Lennart Steinke

Spiele-programmierung

Mehr als 750 Seiten + CD-ROM
€ 16,95 (D)

DAS bhv TASCHEN BUCH

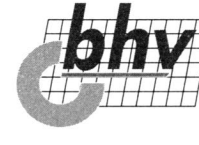
Spieleprogrammierung

Das bhv Taschenbuch

Lennart Steinke

Spieleprogrammierung

Das bhv Taschenbuch



Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen sowie Material auf dem beiliegenden Datenträger wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verleger und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.

Die gewerbliche Nutzung der in diesem Buch und auf der beiliegenden CD gezeigten Modelle und Arbeiten ist nicht zulässig.

Dieses Buch wurde der Umwelt zuliebe auf chlorfrei gebleichtem Papier gedruckt.

Copyright © 2003 by
verlag moderne industrie
Buch AG & Co. KG, Landsberg
Königswinterer Str. 418
D – 53227 Bonn
www.vmi-Buch.de

1. Auflage

07 06 05 04
10 9 8 7 6 5 4 3 2

ISBN 3-8266-8075-8

Printed in Italy

Inhaltsverzeichnis

Vorwort 17

Teil I: Der Design-Prozess 19

1 **Spiele** 21
Was sind Spiele? 21
Arten von Spielen 22
Arten von Spielern 29
Der Mix macht's 31

2 **Entwicklung von Ideen** 35
Ideen finden 35
Kreativitätstechniken 38
Die 6W-Technik 38
Brainstorming 40
Das 6-Farben-Denken 41
Die Walt-Disney-Strategie 42
Das Pferd von hinten aufzäumen 43
Der morphologische Kasten 43
Osbournes Checkliste 44
Mythen, Sagen und Märchen als Inspiration 45
Nordische Sagen als Inspiration 49

3 **Geschichten schreiben** 53
Die Heldenreise 53
Die normale Welt 53
Der Ruf des Abenteuers 54
Die Weigerung 54
Überschreiten der ersten Grenze 54
Die Prüfungen 55
Treffen mit der Göttin/Gefährten 55
Weitere Prüfungen und Verlust des Mentors 55
Der Sieg 55
Die Rückkehr 56
Die Heldenreise angewendet 56
Welten erschaffen 58
Magie 62
Priester und göttliche Magie 62
Drachen 64
Zusammenfassung 64
Lassen Sie den Spieler spielen 65

4 Spielbarkeit und Balance **67**
Gleichgewicht der Kräfte 67
Ausgleichen von Disbalancen 69
Level Design 72
Angemessene Level 73
Das Spiel im Fluss halten 74
Puzzles und Rätsel 75
Arten von Puzzles 76
Physikalische Puzzles 76
Abstrakte Rätsel und Puzzles 82
Logische Rätsel 83
Minispiele 85
Casino- und Glücksspiele 85
Sportspiele 86
Quizspiele 87

5 Rollenspieldesign **89**
Was sind Rollenspiele? 89
Das Rollenspiel-Genre 90
Rollenspiel-System entwerfen 91
Attribute und Fertigkeiten 94
Fähigkeitsbäume 95
Fertigkeitsnetze 96
Den Charakter verbessern – Level up 97
Das aufgabenorientierte System 98
Erfahrungspunkte-System 99
Charakter-Archetypen 101
Charakter-Generierung 103
Bestehende Rollenspielsysteme 104
Ein eigenes System erschaffen 105
Kampf und Magie 105
Kampf 105
Magie 107
Das System erweitern 110
Kampfablauf 112
Echtzeitkämpfe 112
Rundenbasierte Kämpfe 114
Active Time Battles (ATB) 115
Handel und Ökonomie 116

6 Das Design-Dokument 121
Die Essenz des Spieles 121
Warum Sie ein Design-Dokument brauchen 121
Elemente des Design-Dokuments 122
Die Übersicht 123
Die Benutzerschnittstelle 123
Die Personen 124
Schnelldurchlauf 125
Beschreibung der Klassen/Rassen und Berufe 125
Erfahrungspunkte und Stufenanstieg 126
Kampfsystem 126
Magiesystem 127
Künstliche Intelligenz 127
Erweiterungen des Design-Dokuments 128

Teil II: Easy Coding 129

7 Grundlagen der Programmierung 131
Compiler, Linker und was man sonst noch alles braucht 131
Präprozessor 132
Compiler 133
Linker 134
Modularisierung von Programmen 135
Make 137
Wie funktioniert Make? 137
Variablen 138
Muster-Regeln 139
Zeiger 140
*, & und -> 140
Zeiger und Arrays 141
C Strings 144
Speicher anlegen und freigeben 146

8 Von C zu C++ 149
Funktionen in C++ 149
Strukturen in C++ 152
new und delete 153
Objektorientierte Programmierung 154
Hat-Ein und Ist-Ein 157
Polymorphie 158
C++ Gotchas 162

9 Die Standard Template Library 171
Schnelligkeit der STL 171
Grundlagen 171
Parameterübergabe 172
Container 172
Iteratoren 173
Algorithmen 174
Vektoren 174
Bildliches Beispiel für Vektoren 176
Einsatz von Vektoren 177
Listen 178
Bildliches Beispiel für eine Liste 181
Einsatz von Listen 182
Deques und Queues 182
Maps 184
Speichern Sie Zeiger 187
Functors 188
Abschließende Bemerkungen 192

10 Allegro 195
Was ist Allegro? 195
Plattformübergreifende Spiele 196
Funktionalität 196
Grafik 196
Musik und Sound 197
Eingabegeräte 197
Sonstiges 197
Spieleprogrammierung leicht gemacht 198
Alternativen 199
ClanLib 199
DND3E 199
PLib 200
Simple Direct Media Layer (SDL) 200

11 Ein erstes Allegro-Programm 201
Grafikmodi 202
Bilder laden und anzeigen 206
Bewegung 209
Doublebuffering 212
Sound 213
Konstante Geschwindigkeit 215

12 Arten der Interaktion 221
Arten der Interaktion 221
Direkte Kontrolle 222
Mittelbare Kontrolle 222
Indirekte Kontrolle 223
Tastatur 223
Ein Joypad mit 101 Tasten 224
Drücken und Loslassen einer Taste 227
Der Tastaturpuffer 229
Tastatur-Beispielprogramm 231
Die Maus 235
Mauszeiger ändern 237
Mausknöpfe und Räder 241
Der Joystick 243
Joystick-Support aktivieren 244
Das Joystick-Array 244
Abfrage der Buttons 247
Abfrage der Joystick-Richtungen 247
Testprogramm 248
Ein virtuelles Joypad 253
Testprogramm 260

13 Ein erstes Spiel 261
Am Anfang steht die Planung 261
Das Design-Dokument 261
Schritt 1: Der Schlüsselsatz 262
Schritt 2: Eine genauere Spielbeschreibung 262
Schritt 3: Planung der einzelnen Bildschirme 262
Umsetzen des Konzepts 265
Das Makefile 270
Das Titelbild 271
Das Spiel 274
Spielablauf 275
Aufräumarbeiten 282
Erweiterungsmöglichkeiten 283
Spiel als Minispiel benutzen 283

14 Textausgabe 285
Funktionen zur Textausgabe 285
Die textout()-Funktion 285
Die textprintf()-Funktionen 287

Textcodierung 289
Andere Fonts benutzen 290
TTF2PCX 291
Grabber 292
Einen eigenen Font laden 295
Zeichensätze in Farbe 296

15 Menüs und Dialoge 299
Gestaltung von Menüs und Dialogen 299
Farben 300
Größe 300
Marker 301
Animation 301
Sprachausgabe 302
Mausunterstützung 302
Implementierung eines Menüs 303
Menülisten 303
Allegros GUI-Routinen 314
DLG – a Dialog Editor 315
Aufbau der GUI-Routinen 316
Einige Dialog-Prozeduren im Überblick 318
Schönere Buttons 320
Zustände eines GUI-Elements 321

16 Das Quizspiel 329
Planung 329
Spielablauf 330
Benutzerführung 331
Die einzelnen Screens 331
Hauptmenü 331
Die Bestenliste 331
Information 332
Quiz-Screen 332
Eintrag in die Bestenliste 333
Soundeffekte 333
Implementierung 335
Der Menü-Screen 335
Die Bestenliste und der Info-Screen 337
Der Quiz-Screen 337
Das Hauptprogramm 339
Die Button-Prozedur 339

Das Hauptmenü 341
Das Spiel 342
Laden der Fragen 342
Anzeige der Fragen 348
Spielablauf 350
Die Bestenliste 353
Die Programm-Informationen 358
Übungen 359

17 Sprites 361
Arten von Sprites 361
Normale Bitmaps 361
RLE-Sprites 367
Compiled Sprites 368
Sprites erzeugen 368
Animation 378

18 Kollisionsabfragen 389
Treffen sich zwei Sprites, sagt das eine ... 389
Bounding Rectangles 390
Genauere Kollisionen 393

19 Scrolling 403
Virtuelle Kamera 405
Parallax Scrolling 415
Abschließende Bemerkungen 425

20 Soundeffekte 427
Geräusche selbst aufnehmen 428
Kampfgeräusche 429
Bewegung 429
Umgebungsgeräusche 430
Geräusche verändern 430
Sounds/Musik selbst erzeugen 433
Die BeatBox 439

21 Titelbilder und Grafiken 445
Perspektive 445
Draufsicht 446
Seitenansicht 447
Klapp-Perspektive 448
Kavalierperspektive 449
Isometrische Perspektive 449

Zentralperspektive 450
Vogelperspektive 451
Zwei-Fluchtpunktperspektive 451
Entwurf von Titelbildern 452
Drei Helden sollt Ihr sein 452
Kämpfer und Magier 453
Der Gegner im Hintergrund 454
Die Heldensaga 455
Der einzelne Held in Bewegung 455
Das Action-Spiel 456
Weitere Ideen 457
Farben 458
Farbtemperatur 458
Assoziationen mit Farben 459
Internationale Bedeutung von Farben 460

22 Das Veröffentlichen von Spielen 461
Checkliste 461
Das Installationsprogramm 463
Die Webseite Ihres Spieles 468
Bekanntmachung des Spieles 469

Teil III: Rollenspiele 471

23 Tile Based Maps 473
Rogue 473
Kacheln 474
Die Map-Klasse 479
Zusammenfassung 482
Übungen 482

24 Anzeigen von Spielfiguren 485
Sprites in einer Welt aus Kacheln 485
Die Sprite-Klasse 486
Nicht mit dem Kopf durch die Wand 495

25 Komplexere Karten 499
Größere Karten 499
Editoren für Karten 505
Mappy 505
Die Karte laden 512
Mehrere Ebenen 514
Mappy für Karten mit mehreren Ebenen 516

26 Objekte 523
Gegenstände aufnehmen 523
Vom Sprite zum Charakter 524
Verschiedene Objekte 532
Der Objekttyp 539

27 Gegenspieler 543
Monster sind auch nur Sprites 543
Eigenschaften der Monster 544
Die game.ini erweitern 549
Die Monster laden 550
Künstliche Intelligenz 555

28 Kampf 571
Kampfregeln 571
Attacke! 573
Trefferfeedback 580
Gesundheitsanzeige 584

29 Magie und Spezialeffekte 587
Abrakadabra 587
Mana 587
Magische Geschosse 589
Magische Objekte 596
Animierte Kacheln 599
Level wechseln 603
»I need an Exit!« 608

30 Unterschiedliche Charaktere und Levelanstieg 611
Krieger, Magier und Bogenschütze 611
Der Krieger 611
Der Magier 613
Der Bogenschütze 615
Weitere Klassen 617
Der Panzer 617
Der Blutmagier 617
Der Vampir 617
Der Priester 617
Der Berserker 618
Der Gestaltwandler 618
Der Schurke 618
Alternatives Verbesserungssystem 619
Erfahrungspunkte und Stufengrenzen 619

Teil IV: Weiterführende Methoden 623

31 Isometrische Karten 625
Grundlagen der Isometrie 625
Implementierung 628
Alternative Zeichenroutine 633
Bildschirm zu Karte 636

32 Effekte 641
Partikelsysteme 641
Sternensysteme I 641
Sternensysteme II 645
Raketentriebwerke 647
Explosionen 650
Licht- und Transparenzeffekte 653
Allegros Halbtransparenzfunktionen 653
Lichteffekte 656
FBlend – Überblenden auf die schnelle Art 659

33 Seitenübergänge 661
Ein- und Ausblenden 663
StripeFade 664
SymmetricStripeFade 666
StretchFade 667
ScrollFade 668
DoorFade 669
BlindFade 671
Testprogramm 672
Crossfades 675
Alphablending 677
Mosaic 678
DoorCrossFade 680
DoorZoomCrossFade 681
RollOut 683
Weitere Effekte 685

34 Wegfindung mit A* 687
Wohin des Weges 687
Die Idee hinter A* 687
Pseudocode 688
Kosten 689

A* beobachten 689
Implementierung 695
Der Knoten 695
Die Suche 697
Das Hauptprogramm 701

35 Scripting mit Lua 707
Lua downloaden und installieren 707
Lua-Scripte ausführen 707
Lua-Standardbibliotheken 708
Script-Dateien 710
Integration mit eigenem Code 711
Luas Datentypen 711
Der Stack 711
Aufrufen von Lua-Funktionen aus C 713

36 Stimmen der Entwickler 717
Kenny »Sirocco« Thornton 717
Paul Pridham 727
Ben Davis 733
Richard Phipps 739
Robert »Bob« Ohannessian 744
Johan Peitz 747

Teil V: Anhang 753

A Allegro installieren 755
Benötigte Files 755
Installation 756

B Glossar 757

C Inhalt der Buch-CD 761

Index 763

Vorwort

Warum eigene Computerspiele entwickeln? Die Antwort ist eigentlich ganz simpel: Weil man es kann!

Eine gute Idee, ein paar Regeln, ein paar Grafiken und Sounds und schon hat man etwas erschaffen, was mehr ist als die Summe seiner Einzelteile. Zugegeben, ganz so einfach ist es leider in der Praxis doch nicht immer. Potentielle Ideen und Geschichten für Spiele gibt es sicherlich wie Sand am Meer, aber sie müssen einem halt auch einfallen. Zudem lassen sich nicht alle Ideen, die einem im Kopf rumspuken, so einfach umsetzen. Es gibt also bei der Spieleprogrammierung doch so einige Klippen, die umschifft werden müssen, bevor man stolz sein eigenes Spiel präsentieren kann. Dabei soll Ihnen dieses Buch helfen.

Im ersten Teil geht es um den Design-Prozess. Sie erfahren etwas über Ideenfindung, Kreativitätstechniken, Spielbalance und das Erstellen von Storylines für Spiele. Im zweiten Teil lernen Sie die Grundlagen der Spieleentwicklung anhand von konkreten Beispielen kennen. Im dritten Teil wird ein Actionrollenspiel entwickelt, wobei sich die dort angewendeten Techniken sehr leicht auf andere Genres wie Hüpf- und Ballerspiele übertragen lassen. Im vierten Teil wird auf komplexere Themen wie Wegfindung und Skriptsprachen eingegangen. Der fünfte Teil liefert schließlich noch einen kurzen Überblick über die Installation von Allegro, ein Glossar der wichtigsten Begriffe rund um das Thema Spieleprogrammierung sowie Informationen zum Inhalt der Buch-CD.

Alle Programmbeispiele im Buch sind in C beziehungsweise C++ geschrieben. Als Compiler wird in erster Linie die GNU Compiler Collection (GCC) verwendet – alle Programme sollten sich aber ohne Probleme auch mit anderen Compilern wie zum Beispiel Visual C++ übersetzen lassen. Eine Version des GCC für Windows ist auf der CD zum Buch enthalten, ebenso wie die aktuelle Version von Allegro.

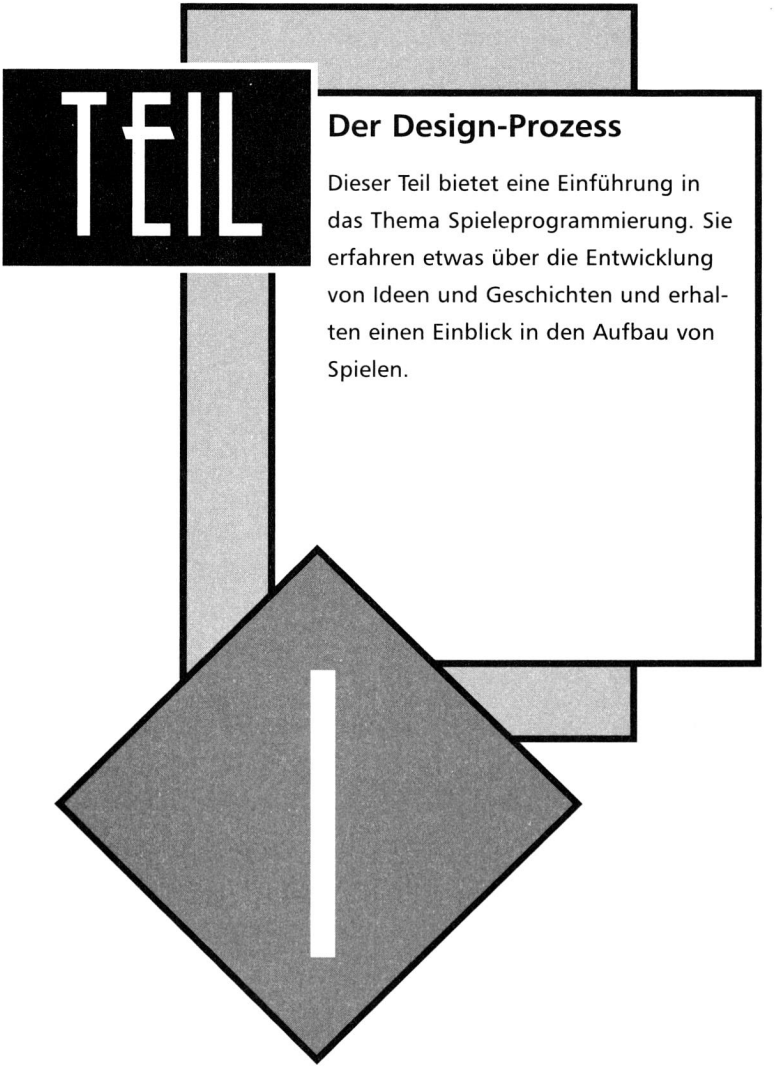
Allegro ist eine plattformübergreifende Bibliothek zur Erstellung von Spielen, d.h., der gleiche Quellcode kann benutzt werden, um Spiele für Windows, LINUX und MacOS zu erstellen. Mit Hilfe von Allegro können Sie sich voll und ganz auf das Entwickeln Ihres Spiels konzentrieren und müssen sich nicht um die langweiligen Details kümmern (falls Sie

sich jedoch für genau diese Details interessieren, können Sie den Allegro-Quellcode komplett einsehen und ggf. auch verändern).

Weitere Vorteile von Allegro sind die gute Dokumentation, die gewaltige Menge an Beispielen und die sehr hilfsbereite Onlinegemeinschaft auf *http://www.allegro.cc/*. Falls Sie also einmal eine Frage haben, dann zögern Sie nicht, diese in dem entsprechenden Forum auf *Allegro.cc* zu stellen. Sie können dort auch Ihre fertigen Spiele präsentieren und Erfahrungen mit anderen Entwicklern austauschen.

Ich wünsche Ihnen viel Spaß beim Lesen dieses Buches und freue mich schon darauf, bald ein von Ihnen erstelltes Spiel testen und spielen zu können.

Lennart Steinke



Der Design-Prozess

Dieser Teil bietet eine Einführung in das Thema Spieleprogrammierung. Sie erfahren etwas über die Entwicklung von Ideen und Geschichten und erhalten einen Einblick in den Aufbau von Spielen.

1 Spiele

Spiele faszinieren die Menschen schon seit Jahrhunderten. Im Britischen Museum steht ein Brettspiel, das weit über 4600 Jahre alt ist. Dieses Spiel, welches in den königlichen Grabmälern von Ur in Mesopotamien (der heutige Irak) gefunden wurde, ist jedoch bei weitem nicht das älteste. Bei Ausgrabungen in Ägypten wurden mehrere Dutzend Senetbretter entdeckt – das älteste ist über 5000 Jahre alt. In der Tat stellt der griechische Philosoph Platon Spiele in seinem Werk »Phaidros« als die Erfindung des ägyptischen Gottes Thot dar, ein Hinweis darauf, dass die ältesten Spiele aus Ägypten zu kommen scheinen. Das Senet-Spiel gilt übrigens als einer der Vorläufer des heutigen Backgammon.

Was sind Spiele?

Spiele basieren auf einem Satz von Regeln, fordern den Spieler heraus und haben entweder eine Gewinn- oder Verlustbedingung. Das Spiel definiert im gewissen Sinne eine eigene Welt, die nur der Unterhaltung des Spielers oder der Spieler dient.

Alles, was dieser Beschreibung entspricht, ist ein Spiel. Wenn jemand einfach nur schnell rennt, dann ist dies zwar eine sportliche Leistung, aber noch kein Spiel. Fügt man jedoch Regeln hinzu (Starte hier, renne bis zu dieser Linie, sobald das Startsignal ertönt ist.) und definiert eine Siegbedingung (Wer die Strecke in der kürzesten Zeit hinter sich bringt, gewinnt.), dann haben wir ein Spiel.

Gibt man einem Kind eine Autorennbahn, dann ist dies in erster Linie ein Spielzeug. Das Kind kann die Rennbahn benutzen, um Spiele im Sinne der obigen Definition zu spielen. Es kann aber einfach auch nur Spaß daran haben, mit den Autos im Kreis zu fahren und sie möglichst spektakulär aus den Kurven fliegen zu lassen.

Einige der erfolgreicheren PC-Spiele-Titel der letzten Zeit sind nach der obigen Definition eher ein Spielzeug als ein Spiel. Eine Stadt zu planen, zu bauen und zu verwalten kann sehr viel Spaß bereiten und für eine lang anhaltende Motivation sorgen. Doch fehlt häufig die Siegbedingung. Der Spieler wird immer noch etwas finden, das es zu verbessern gilt.

Auch virtuelle Haustiere und ihre Hightech-Nachfolger wie die »Sims« haben meist keine eng umrissenen Siegbedingungen. Man spielt diese

Spiele um des Spielens willen – oder besser: Man spielt mit diesem Spielzeug um des Spielens willen.

Arten von Spielen

Es gibt verschiedene Methoden, Spiele zu kategorisieren. Bei Spielen im »echten Leben« (im Gegensatz zur virtuellen Realität des Computers) unterscheidet man zwischen Sportspielen, Brettspielen, Geschicklichkeitsspielen, Kartenspielen, Reaktionsspielen, Quizspielen und Erzählspielen. All diese Spiele gibt es auch als Computer-Variante. Zusätzlich zu diesen Kategorien gibt es am Computer auch noch Ballerspiele, Hüpfspiele und Strategiespiele.

Andere katalogisieren Computerspiele eher wie Bücher und unterteilen sie in Fantasy-, Science-Fiction-, Mantel-und-Degen-, Krimi- und Real-Setting Spiele.

Und wie man es auch einteilt, es wird immer das eine oder andere Spiel geben, das aus dem Raster rutscht. Und das ist auch gut so. Es gibt Fantasy-Spiele, die in einer Science-Fiction-Umgebung spielen. Es gibt Krimis, die in allen möglichen Zeiten von der Vergangenheit bis zur fernen Zukunft angesiedelt sind. In manchen Spielen reist man durch die Zeit, in anderen über die Welt.

Versuchen Sie nicht, in engen vorgegebenen Bahnen zu denken, wenn Sie ein Spiel entwickeln. Lassen Sie sich nicht durch Genres einschränken und überlassen Sie es den Spielern und Spiele-Testern Ihr Spiel einzuordnen. Und das ist beileibe keine einfache Aufgabe.

Um eine gemeinsame Sprache zu schaffen, werde ich kurz die bekanntesten Genres definieren. Lassen Sie sich aber nicht zu dem Glauben verleiten, dass diese Definitionen allgemein gültig sind. Jeder Spiele-Designer und jeder Spiele-Entwickler hat wahrscheinlich andere Definitionen (oder sogar eine andere Liste der grundlegenden Genres).

Action

Actionspiele legen viel Wert auf die Reaktionsfähigkeit des Spielers. Gute Reflexe, Timing und ein schneller Finger auf dem Feuerknopf machen hier einen Großteil des Spieles aus. Actionspiele fanden ihren Anfang bei »Space War«, dem ersten Computerspiel überhaupt. Dann folgten unzählige Varianten von Spielen, in denen böse Außerirdische daran gehindert

werden mussten, die Erde zu erobern, und heutzutage sind die First-Person-Shooter die Glanzlichter dieses Genres.

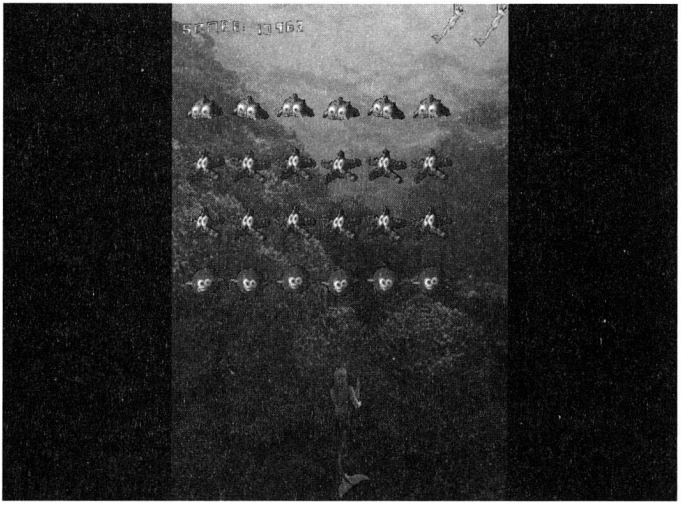


Abbildung 1.1: Ein Actionspiel

Geschicklichkeit

Geschicklichkeitsspiele erfordern Timing und gute Hand-Augen-Koordination. Hüpfspiele sind der klassische Vertreter dieser Kategorie (siehe Abbildung 1.2).

Abenteuer

Abenteuerspiele erzählen eine Geschichte. Der Spieler kämpft sich durch die einzelnen Handlungsstränge, löst Rätsel und erlebt Abenteuer jeglicher Art und Weise. Neben den klassischen Abenteuerspielen wie Adventure- und Lucas-Arts-Spielen wie der »Monkey Island« und »Maniac Mansion« Reihe gehören auch die meisten Rollenspiele in diese Kategorie (siehe Abbildung 1.3).

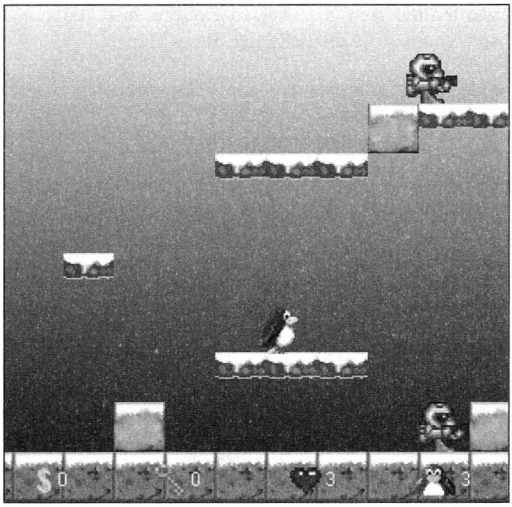


Abbildung 1.2: Ein Hüpfspiel

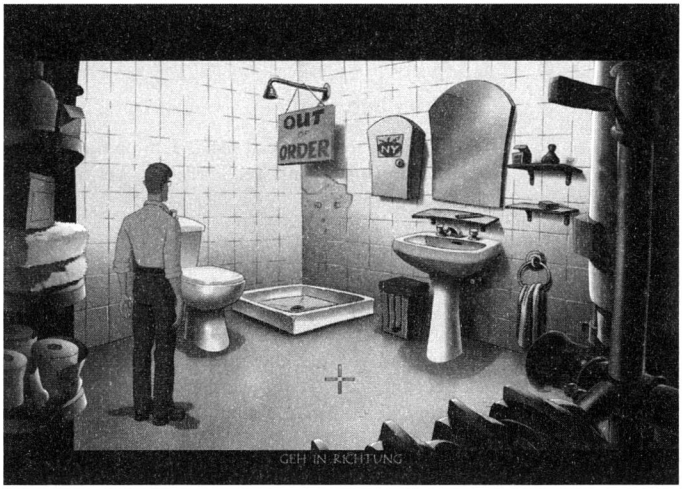


Abbildung 1.3: Ein Abenteuerspiel (»Runaway: A Road Adventure«)

Strategie

Strategiespiele erfordern Planung. Der Spieler muss seine Einheiten geschickt einsetzen, um sich gegen seine Widersacher durchzusetzen. Die klassischen Strategiespiele, die eher an militärische Planspiele erinnern, sind inzwischen durch Echtzeitstrategiespiele verdrängt worden.

Abbildung 1.4: Ein Echtzeitstrategiespiel (WarCraft III)

Echtzeitstrategiespiele (RTS, ***R**eal **T**ime **S**trategy*) sind seit den Tagen von Dune 2 aus der Spiellandschaft nicht mehr wegzudenken. Zwar haben diese Spiele auch einen Simulations- und Actiongehalt, aber der Schwerpunkt liegt noch immer klar bei der strategischen Planung.

Simulation

Bei Simulationen steht die möglichst realistische Nachbildung der Realität im Vordergrund. Simulationen gibt es in vielen Formen. Vom traditionellen Flugsimulator über Simulatoren von Modeleisenbahnen über Züge und Straßenbahnen bis hin zur Simulation von Aufzügen in Hochhäusern reicht hier die Palette.

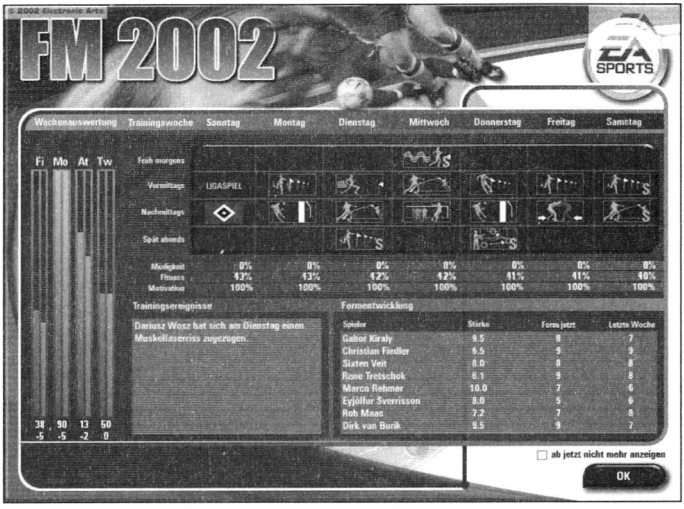


Abbildung 1.5: Eine Fußball-Management-Simulation (»FM 2002«)

Die meisten Spiele brauchen eine Simulationskomponente, und sei es nur, um die Spielphysik nachzubilden. Rollenspiele werden zum Beispiel auch häufig als Heldensimulatoren bezeichnet.

Denkspiele

Sobald das Lösen kniffliger Probleme im Vordergrund steht, hat man ein Denkspiel vor sich. Die Liste dieser Spiele ist recht lang. Von Lemmings über Spiele wie Push-Over bis hin zum innovativen Incredible Machine – Denkspiele kommen in vielen Varianten (siehe Abbildung 1.6).

Spielzeug

Spiele geben ein Ziel vor, Spielzeug erlaubt es dem Spieler, sich seine Ziele selber zu stecken. Die bekannteren Vertreter dieses Genres sind die verschiedenen virtuellen Haustiere und der Klassiker »Little Computer Research Project« (siehe Abbildung 1.7).

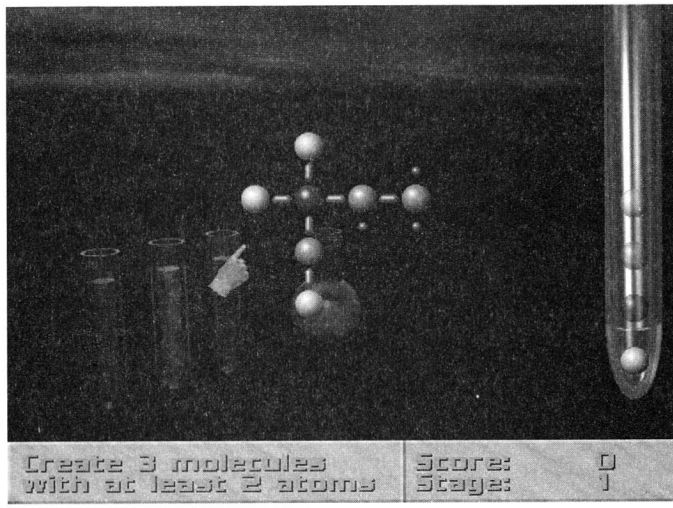


Abbildung 1.6: Molecule, ein Puzzle / Denkspiel

Abbildung 1.7: Die Sims

Ein Rollenspiel, in dem der Spieler wirklich alle Freiheiten hat, kann vom Spiel zum Spielzeug werden.

Sportspiele

Es begann alles mit Pong, und inzwischen gibt es fast keine bestehende oder fiktive Sportart, die noch nicht zu einem Spiel verarbeitet wurde. Tennis, Eishockey, Turmspringen, Springreiten, Fußball, Radfahren, Schwimmen, Skateboard fahren und Turnen – all dies gibt es inzwischen in der Form von Computerspielen.

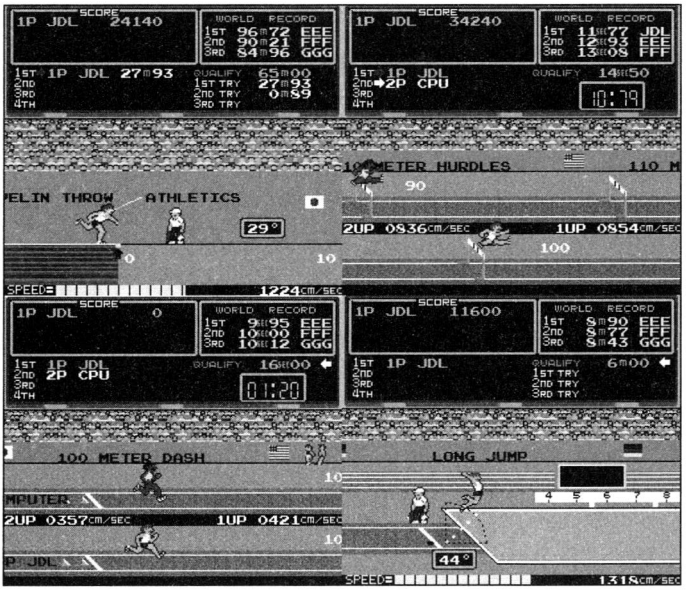


Abbildung 1.8: Das klassische Sportspiel (»Track and Field«)

Kombinationen

Sportspiele haben meist einen hohen Anteil an Action- und Geschicklichkeitselementen, Abenteuerspiele enthalten häufig Puzzles und Rätsel,

die eigentlich eher zum Denkspielsegment gehören. Dadurch kann man dem Spieler mehr Abwechslung bieten.

Arten von Spielern

So wie es verschiedene Arten von Spielen gibt, gibt es auch verschiedene Arten von Spielern. Einige wollen nur etwas Zeit mit einem Spiel verbringen, andere stellen Strategiepläne zum Gewinnen des Spieles zusammen. Lernen Sie die einzelnen Spielertypen kennen, und nutzen Sie dieses Wissen um jedem dieser Typen etwas zu bieten.

Der Power Spieler

Der Power Gamer kennt das Spiel in- und auswendig. Er kennt jede Regel, und was noch wichtiger ist: Er kennt auch jedes Schlupfloch in den Regeln. Wenn es einen Fehler im Spiel gibt, den man zu Gunsten des Spielers ausnutzen kann: Er wird ihn finden.

Der Power Gamer sucht den kürzesten Weg zur Meisterschaft des Spieles. Die Story des Spieles ist ihm größtenteils egal, er will nur die beste Ausrüstung, die stärksten Waffen und die meisten Punkte.

Der Hardcore Spieler

Der Hardcore Gamer ist dem Power Gamer sehr ähnlich. Auch er strebt nach dem »perfekten Spiel«. Aber im Gegensatz zum Power Spieler versucht er dabei nicht zu schummeln. Er investiert eine Menge Zeit in ein Spiel und sucht die Informationen zusammen, die dann später von den Power Gamern benutzt werden, um sich schnell nach oben zu mogeln. Je schwerer das Spiel um so besser für ihn – er sucht die Herausforderung.

Der Gelegenheitsspieler

Der Gelegenheitsspieler will nur seine Zeit totschlagen. Ein Spiel das Gelegenheitsspieler ansprechen soll muss leicht zugänglich sein. Die Grundregeln sollten intuitiv erfassbar sein, und die Bedienung leicht zugänglich. Wenn Sie es schaffen, Gelegenheitsspieler in Ihren Bann zu ziehen, dann haben Sie es geschafft: Ihr Spiel wird ein Erfolg.

Diese Spielertypen kommen wieder in verschiedenen Ausprägungen.

Der Kämpfer

Der Kämpfer bevorzugt eine einfache, direkte Lösung für die auftretenden Probleme. Diese Lösung besteht meist aus einem Schwert, Feuerball oder Raketenwerfer. Diese Art von Spieler versucht einfach aus Prinzip mal auf alles einzuschlagen, was die Welt zu bieten hat, und ist enttäuscht, wenn dabei nichts kaputt geht.

Diese Art von Spieler wird versuchen Meteore im Tempel herbeizuzaubern, nur um zu sehen was passiert. Sie werden mit Pfeilen auf Fenster schießen und erwarten, dass das Glas zerbricht. Um ihn glücklich zu machen, braucht es nur jede Menge Waffen und Gegner.

Der Rätselfuchs

Der Rätselfuchs ist das Gegenteil des Kämpfers. Er liebt die logischen Herausforderungen. Puzzle und Knobeleien sind die liebsten Elemente des Rätselknackers. Je höher der Schwierigkeitsgrad, umso besser. Er kann sich über längere Zeit hinweg mit dem Lösen eines kniffeligen Problems beschäftigen. Rätsel, die für die meisten zu schwer und frustrierend sind, spornen ihn an. Geben Sie ihm Rätsel und die Möglichkeit Gegenstände zu kombinieren und eventuell unkonventionell einzusetzen. Wenn der Kämpfer der B.A. Barracus unter den Spielern ist, dann ist der Rätselfuchs McGuyver.

Der Sammler

Der Sammler liebt es, alle Schätze und Geheimnisse des Spieles zu finden. Er wird nicht eher ruhen, bis er nicht auch das letzte Geheimnis gefunden hat. Er will mit den Gegenständen, die er gefunden hat, handeln und am besten selbst eigene (am besten magische) Gegenstände herstellen können. Um ihn glücklich zu machen, braucht das Spiel eine komplexe Ökonomie und eine Vielzahl an Gegenständen, die es zu entdecken gilt.

Der Entdecker

Der Entdecker erforscht jeden Winkel der Spielwelt. Und er erwartet in diesen Winkeln auch interessante Dinge zu finden und eventuell sogar neue Facetten der Hintergrundgeschichte aufzudecken. Um ein Spiel zu entwickeln, das Entdecker anspricht, braucht es große, möglichst ausgearbeitete Spielwelten.

Der Mix macht's

Nun kennen Sie die grundlegenden Spielformen und die verschiedenen Spielertypen. Jetzt müssen Sie all dies nur noch so kombinieren, dass sich der Spieler nicht mehr von dem Spiel lösen kann, sondern am liebsten konstant spielen würde.

Es gibt dafür kein Rezept, aber es ist immer eine gute Idee, möglichst viele Komponenten im Spiel zu haben. Dies erlaubt es dem Spieler auch eine Entwicklung zu durchleben. So könnte er als Entdecker anfangen, sich mit der Zeit mit den Regeln vertraut machen und dann zu einem Hardcore Spieler werden. Oder ein Kämpfer beginnt mit der Zeit immer mehr Gegenstände zu sammeln und wird zu einem Sammler.

Kämpfer ansprechen

Neben Kämpfen und vielen Dingen, die man zu Kleinholz verarbeiten kann, kann man den Kämpfer auch auf andere Weise locken: Und zwar über seinen Stolz. Stellen Sie sich vor, dass er durch seine vielen erfolgreichen Kämpfe einen gewissen Ruf bekommt. Dies könnte sich zum Beispiel durch seinen Titel ausdrücken – »Gnorg der Starke« klingt doch schon recht gut, aber »Gnorg der Mächtige, Schwertmeister von Karuthan« ist doch noch eindrucksvoller. Geben Sie dem Kämpfer Minispiele, durch die er sich mit anderen Spielern vergleichen kann. So könnte es einen Wettstreit gegen die Uhr geben, in dem man sich einer festgelegten Anzahl von Gegnern erwehren muss. Die dabei erreichte Zeit kann nun benutzt werden, um die eigene Leistung mit der der anderen zu vergleichen. Richten Sie auf ihrer Webseite ein Forum ein, und rufen Sie zu Wettbewerben auf um den besten Spieler in einem bestimmten Minispiel zu ermitteln.

Sammler ansprechen

Sie brauchen einen großen Variantenreichtum, um Sammler anzusprechen. Und die besten Stücke müssen sehr rar sein. Streuen Sie im Spiel Gerüchte über wundersame Waffen, die man in entlegenen Winkeln finden kann – wenn man den Mut hat es zu probieren.

Erlauben Sie es dem Spieler seine mitgeführten Gegenstände, sein Inventar (oder engl. Inventory) selbst zu verwalten. Wenn in Ihrem System verschiedene Gegenstände unterschiedliche Größen haben können (und damit auch verschieden viel Platz in der Übersicht einnehmen), dann wird

der Spieler früher oder später vor die Aufgabe gestellt, seinen Bestand neu ordnen zu müssen, oder sich zwischen den bereits gefundenen Gegenständen und gerade errungenen zu entscheiden.

Wenn der Spieler eigene Gegenstände erschaffen kann, haben Sie den Sammler normalerweise ganz in den Bann gezogen. Ein mächtiges Schwert zu finden ist eine tolle Sache. Aber sich selbst eines zu erschaffen ist noch viel besser. Der schöpferische Akt schafft eine Verbindung zwischen dem Gegenstand und dem Erschaffer. Es ist nicht mehr nur ein magischer Gegenstand. Nein, es ist der eigene, selbst geschaffene magische Gegenstand.

Ein Spiel für Rätselfüchse

Ein Spiel für Rätselfüchse braucht natürlich eine hohe Anzahl von Rätseln und logischen Puzzles. Und diese müssen erst mal erdacht und in das Spiel eingebaut werden. Das Problem bei Rätseln ist meistens, dass Sie deutlich an Reiz verlieren, sobald sie einmal gelöst worden sind.

Wenn Sie an einigen Stellen anstatt von festen Rätseln dynamische erstellte Rätselaufgaben stellen, können Sie die Langzeit-Motivation wieder steigern. Ein Beispiel: Im Dorf Hagenhausen ist ein Mord geschehen. Der Spieler soll den Täter finden (bzw. die eigene Unschuld beweisen). Anstatt nun Täter und Beweise festzulegen, bestimmen Sie bei jedem Spiel Täter, Opfer und Tatwaffe zufällig. Natürlich müssen dann auch die Hinweise auf den Täter entsprechend gewählt werden. Auf diese Weise kann der Spieler jedes Mal ein neues Problem knacken.

Neben den klassischen Rätseln können Sie den Grübler auch mit kombinatorischen Aufgaben fesseln. Während der Sammler sein Glück beim Erschaffen neuer Gegenstände versucht, wird der Tüftler versuchen zu verstehen, auf welche Weise die Kombinationen zu den Ergebnissen führen.

Ein Spiel für Entdecker

Entdecker bevorzugen große, lebendige Welten. Welten, in denen etwas passiert. Welten, in denen sie zwar die Dinge beeinflussen können, aber die Umgebung noch immer eine Eigendynamik hat.

Entdecker werden Ihnen all die kleinen Details im Spiel danken. Die Dinge, von denen Sie dachten, dass sie wohl nie jemand bemerken wird - der Entdecker wird sie finden.

Um Entdecker anzusprechen, sollten Sie viele versteckte Bereiche in das Spiel integrieren, damit es auch etwas zu finden gibt. Oder zeigen Sie ihnen Orte, die es zwar gibt, zu denen sie aber bis jetzt noch keinen Zugang haben. Die bekannteste Methode dafür ist die Schatztruhe auf einer Insel, an die der Held nicht herankommt bis er Schwimmen gelernt hat.

Machen Sie die Welt lebendig. Wenn der Spieler einer Figur im Spiel das Leben rettet, dann lassen Sie in einem Nachbardorf den Bruder dieser Figur auf den Helden zukommen und sich bedanken.

Versuchen Sie möglichst viele Nebenaufgaben in das Spiel zu integrieren oder bevölkern Sie manche Gegenden einfach zufällig mit Monstern und neuen Höhlen, die der Held erforschen kann. Erlauben Sie es dem Helden selbst zu entscheiden, ob er die Goldene Schale von Riix suchen möchte, oder ob er lieber durch die Wälder streifen und Pilze für einen Zaubertrank sammeln möchte.

Verbindende Elemente

Das Anhäufen von Schätzen und das Sammeln seltener Gegenstände sind Dinge, die jeden Spieler in einem gewissen Maße ansprechen. Sie sollten aus diesem Grund versuchen, eine Sammelkomponente in Ihr Spiel einzubauen. In Hüpfspielen muss man zum Beispiel häufig Buchstaben oder besondere Gegenstände einsammeln, um ein Extraleben zu bekommen. In Ballerspielen hinterlassen die besiegten Raumschiffe manchmal Münzen und extra Waffen, zum Teil gibt es auch einen Hangar, in dem man zwischen den Leveln sein Schiff verbessern kann (wenn man bereit ist, die entsprechende Anzahl an Punkten zu opfern).

Ehrgeiz ist ein weiterer wichtiger Punkt. Geben Sie Ihren Spielern die Möglichkeit sich in vielerlei Hinsicht zu vergleichen. In einem Actionspiel könnte neben der Punktzahl auch die Treffsicherheit, die gebrauchte Zeit und Menge der gefundenen Geheimnisse angezeigt werden. In einem Puzzlespiel sollten Sie auch mehr bewerten als nur die Endpunktzahl. Führen Sie für jeden Aspekt des Spieles High-Score-Listen ein. Warum sollte es in einem Prügelspiel keine Liste der schnellsten Knock-outs geben? Oder den höchsten Schaden, der mit einem einzelnen Angriff ausgeteilt wurde?

Geben Sie Ihren Spielern die Möglichkeit diese Ergebnisse online zu stellen. Stellen Sie sich vor, Sie haben in Ihrem Lieblingsspiel ein recht gutes Ergebnis. Sie vergleichen Ihre Werte online mit denen der anderen Spie-

ler und stellen fest, dass Sie sich im oberen Bereich der Liste befinden. Allerdings stockt Ihnen der Atem, als Sie das Ergebnis des Erstplazierten sehen... es kann doch nicht sein, dass der um so viel besser ist? Also geht es zurück zum Spiel, um herauszufinden man diese Punktezahl schlagen kann.

Ehrgeiz und ein gesundes Rivalitätsgefühl sind das Salz in der Suppe der Spiele. Nutzen Sie sie zu Ihrem Vorteil.

2 Entwicklung von Ideen

Da Sie dieses Buch in Händen halten, haben Sie wahrscheinlich auch schon eine gute Idee für ein Spiel. Eine gute Idee ist der Kern eines jeden Spieles. Welcher Art die Idee ist, ist zweitrangig. Manchmal sieht man einen Film oder liest ein Buch und denkt sich: »Wow... das würde auch sehr gut in ein Spiel passen!« Oder der Blick fällt auf eine bestimmte Grafik und man denkt sich, dass diese Art der Gestaltung sicherlich genial zu einem Spiel passen würde. Manchmal denkt man auch beim Spielen eines existierendes Games »Warum kann ich jetzt nicht einfach [fehlendes Feature]? Wenn das jetzt gehen würde, dann wäre es perfekt!« – und beschließt es besser zu machen.

All diese Ideen können die Grundlage eines guten Spieles werden. Aber Ideen müssen gesammelt, gepflegt und ausgearbeitet werden.

Ideen finden

Ideen für Spiele fliegen einem manchmal zu. Es gibt Zeiten, in denen man jeden Tag eine neue Idee hat. Legen Sie sich ein Notizbuch zu (ja, die altmodischen Dinger aus Papier) und notieren Sie sich alle Ideen. Gehen Sie nicht zu sehr ins Detail, nutzen Sie die Notizen mehr als Erinnerungshilfe. Was aber tun, wenn einem einfach nichts einfallen will? Dann lehnen Sie sich zurück und lassen Sie sich inspirieren.

Bücher

Es gibt Bücher zu beinahe jedem Thema. Gehen Sie mal einen Nachmittag in die öffentliche Bücherei und streifen Sie einfach nur so herum. Werfen Sie einen Blick in Bücher, die Sie sonst meiden würden, werfen Sie einen Blick in Fachbücher über Themen, von denen Sie keine oder kaum eine Ahnung haben. Schlagen Sie Lexika willkürlich auf und stöbern Sie ein bisschen herum.

Wenn Sie ein Fan der klassischen Fantasy-Geschichten sind, dann lesen Sie doch mal einen »Shadowrun«-Roman. Lesen Sie sonst immer ernste und Action-geladene Bücher, könnte ein humorvoller Roman eine willkommene Abwechslung sein.

Film und Fernsehen

Gehen Sie ins Kino und sehen Sie die Kino-Magazine im Fernsehen an. Manchmal können Kleinigkeiten in einem Film der Ausgangspunkt für eine Vielzahl von Ideen sein. Überlegen Sie, wie ein bestimmter Film als Spiel aussehen könnte. Lassen Sie dabei Ihrer Fantasie freien Lauf – wenn Sie die erste Szene als Rollenspiel, die zweite als Flugsimulator und das Ende als Puzzle realisieren würden – nur zu. Sie haben alle Freiheiten. Überlegen Sie sich, wie der Spieler mit dem Spiel interagiert, wo er Möglichkeiten hat, die nicht im Film vorkommen, und wie Sie damit umgehen könnten... lassen Sie Ihre Gedanken treiben und stellen Sie sich vor den Film durchzuspielen.

Stellen Sie sich einmal vor, wie ein schlecht synchronisierter Kung-Fu-Film mit schlechten Schauspielern und an den Haaren herbeigezogener Geschichte als Computerspiel aussehen würde. Insbesondere wenn Sie die Synchronisation als Stilelement in Ihrem Spiel einsetzen wollen.

Wenn Sie eine Herausforderung möchten, dann versuchen Sie doch mal eine Nachrichtensendung zu einem Spiel zu machen. Die einzelnen Nachrichten könnten kleine Spiele sein, die durch den Sprecher verbunden sind. Oder machen Sie doch ein Spiel über die Entstehung der Nachrichten. Und wie sieht es mit der Werbung aus? Können Sie Werbespots in Spiele verwandeln? Einen ganzen Werbeblock?

Natur und Umgebung

Gehen Sie mal raus und schauen Sie sich um. Hatten Sie beim Überqueren der Straße schon mal das Gefühl, dass die Autos nur da sind, um es Ihnen möglichst schwer zu machen? Wenn ja, dann haben Sie jetzt eine Vorstellung, wie der Klassiker Frogger entstanden sein könnte. Die Idee zu Pikmin kam Nintendos Star-Designer Shigeru Miyamoto, als er das Wuseln in seinem Garten betrachtete. Die Schlachten an den Grabbeltischen beim Schlussverkauf sind eine gute Vorlage für ein Echtzeit Strategiespiel. Sie könnten aus der gleichen Vorlage auch ein klassisches Labyrinthspiel im Sinne von Pac-Man machen. Haben Sie schon mal in einer Sommernacht nach Stechmücken in Ihrem Schlafzimmer gesucht? Mit der richtigen Sichtweise können Sie beinahe alles zu einem Spiel machen.

Comics und Zeichentrickserien

Comics und Zeichentrickserien haben sehr viel mit Computerspielen gemeinsam. Schauen Sie sich mal in aller Ruhe ein paar Zeichentrickserien

am Samstagmorgen an, und stöbern Sie mal im lokalen Comicladen. Ihre Videothek sollte auch einige bekanntere Animes (japanische Zeichentrickfilme) zum Ausleihen haben. Schauen Sie sich ein paar an, und achten Sie auf die Art und Weise, wie mit einfachen Mitteln ein Gefühl der Tiefe vermittelt wird. Gerade in Animes wird sehr häufig mit verschiedenen Ebenen gearbeitet, die dann gegeneinander verschoben werden. Diese Art der Animation eignet sich auch sehr gut für Computerspiele. Beim Betrachten vom Zeichentrickfilmen können Sie also neben Anregungen für Spielideen auch einiges über den Aufbau von Animationen lernen.

Ist das nicht eine gute Begründung, um sich Zeichentrickfilme anzusehen?

Diskutieren Sie Spiele

Diskutieren Sie Ihre aktuellen Lieblingsspiele und Demos mit Freunden. Achten Sie auf die Dinge, die besonders hervorgehoben werden. Wenn es um Aspekte des Spieles geht, die Ihren Freunden und Bekannten nicht so gut gefallen haben, versuchen Sie, das Gespräch in Richtung Verbesserungsvorschläge zu leiten. Lesen Sie die Internetforen von Spielen, die Sie interessieren, um die Meinung anderer Spieler zu erfahren. Sie werden überrascht sein, wie viele verschiedene Meinungen zu auch eher unscheinbaren Themen existieren.

Spielen Sie

Spielen Sie. Spielen Sie alle Spiele, die Ihnen interessant erscheinen - egal ob Computer-, Brett-, Sammel- oder Geduldsspiel: Gehen Sie offen darauf zu und versuchen Sie die Stärken und Schwächen der jeweiligen Spielform zu erkunden.

Bei Computer- und Videospielen sollten Sie neben den aktuellen Spieletiteln auch einen Blick auf ältere Software werfen. Mit etwas Glück können Sie auf einem Flohmarkt ein Super Nintendo Entertainment System oder ein Sega Megadrive mit Spielen für ein paar Euro ergattern. Auch auf dem Commodore 64 gab es sehr viele sehr gute Spiele, die mehr als nur einen flüchtigen Blick wert sind. Falls Sie keine Lust verspüren sich durch Unmengen von Spielen zu wühlen, sollten Sie in Ihrer bevorzugten Internet Suchmaschine nach Begriffen wie »top 10 games« in Verbindung mit dem gewünschten System suchen. Die meisten Seiten listen dann neben dem Namen des Spieles auch einen Grund auf, warum dieses Spiel in keiner Sammlung fehlen sollte. Allein diese Beschreibungen können Ihnen schon wieder einige neue Ideen liefern.

Kreativitätstechniken

Kreativitätstechniken helfen Ihnen bestehende Ideen zu vertiefen und auf neue Ideen zu kommen.

Die 6W-Technik

Die 6W-Technik ist eine der grundlegendsten und einfachsten Techniken, die es gibt. Der Name ergibt sich aus den Anfangsbuchstaben der Fragen die gestellt werden: *Wer?, Wann?, Was?, Wo?, Wie?, Warum?*

Diese Technik hilft einem seine Gedanken besser zu ordnen und stellt sicher, dass man auch alle Teilaspekte in Betracht zieht.

Ein Beispiel: Die Grundidee ist »Ein Junger Held gegen die Welt der Monster«. Schreiben Sie einfach alles auf, was Ihnen zu den jeweiligen Fragen in den Sinn kommt. Es geht nicht darum die beste Antwort zu finden, sondern möglichst viele unterschiedliche Ideen zu sammeln. Hier ist eine mögliche Liste von Antworten.

Wer ist der Akteur?

- ✔ Ein heldenhafter Soldat, dessen Auftrag es ist, die Monster zu vernichten.
- ✔ Ein Jugendlicher aus einem Dorf, der zufällig in die Sache verwickelt wird.
- ✔ Jemand, der von seiner Berufung hört, gegen die Mächte der Finsternis anzutreten.
- ✔ Jemand, der sich an den Monstern rächen will, weil Sie jemanden verletzt, getötet oder entführt haben, der dem Held wichtig ist.

Wann – Zu welchem Zeitpunkt findet alles statt?

- ✔ In der Zeit des dunklen Mittelalters – Aberglaube ist weit verbreitet
- ✔ In der Gegenwart
- ✔ In einer mittelalterlichen Fantasy-Welt – mit allem, was dazu gehört: Elfen, Trolle, Einhörner und Magier

- ✔ In einer Variante der Gegenwart, zum Beispiel in einer von Königen regierten Welt, oder einer Welt, in der Magie den Platz der Technik einnimmt
- ✔ Eine nahe Zukunft mit Bio-Implantaten und Konzernen, die alles regulieren
- ✔ Eine Gegenwart, in der alle Verschwörungstheorien der Wahrheit entsprechen
- ✔ Eine ferne Zukunft, entweder die Heile-Welt-Variante à la Star Trek, oder eine düstere Variante à la Aliens

Was – Worum geht es in dem Spiel?

- ✔ Monster metzeln, wer fragt da nach einem Grund?
- ✔ Ein magisches Artefakt finden, um die Monster zu besiegen
- ✔ Ein magisches Artefakt vor den Widersachern finden, damit nicht die Bösen über das Gute siegen
- ✔ Der Klassiker: Den Weltuntergang verhindern
- ✔ Einen Freund retten

Wo – Wo spielt das Spiel

Welche Umgebung kommt vor, welche Klimazonen? Wald? Stadt? Wüste? Höhlen? Ist stark abhängig von der Zeit, aber die klassischen Monster-Umgebungen scheinen eine gute Möglichkeit zu sein.

- ✔ Einsame Dörfer
- ✔ Unheimliche Schlösser
- ✔ Friedhöfe
- ✔ Laboratorien verrückter Professoren
- ✔ Kerker und Verliese
- ✔ Grabmäler (Gruften, Pyramiden)

Wie – Wie erfüllt der Held seine Aufgabe?

- ✔ Wie macht er das, was er tut?
  - ✔ Normales Spiel – voll durch alle Monster durch
  - ✔ Schleichen und Verstecken – Man muss die Gegner einzeln und möglichst leise ausschalten.
- ✔ Welche Hilfsmittel hat er?
  - ✔ Ein magisches Schwert
  - ✔ High–End Monster-Vernichtungsausrüstung, wie Laser und Flammenwerfer
  - ✔ Netze, um Gegner zu fangen
  - ✔ Pistole mit Silberkugeln
  - ✔ Zaubersprüche
  - ✔ Fallen

Warum – Was ist die eigentliche Motivation hinter seinen Taten?

- ✔ Es ist sein Job – der Held ist professioneller Monsterjäger, Soldat oder Polizist.
- ✔ Er will einem Freund helfen / ihn befreien.
- ✔ Es ist seine Bestimmung.
- ✔ Er will Rache, für den Verlust eines Freundes / Verwandten.
- ✔ Er sucht nach einem Schatz, der von den Monstern gehütet wird, oder hinter dem auch die Monster her sind.

Da kommt doch schon eine ganz schöne Liste zusammen. Und Sie können die einzelnen Ideen auch beliebig kombinieren.

Brainstorming

Das Brainstorming ist eine der bekanntesten Methoden. Vom Prinzip her handelt es sich dabei um ein Gespräch, in dem alle Beteiligten ihre Meinung frei äußern können – solange sie nicht die Aussagen der anderen Teilnehmer kommentieren oder gar kritisieren. Es ist wichtig, dass wirk-

lich jede Idee gleichberechtigt aufgenommen wird. Umsetzbarkeit ist nicht wichtig. Je abgedrehter die Idee ist, umso besser. Ausgefallene Ideen helfen den Horizont zu erweitern. Quantität geht vor Qualität: Lassen Sie alle Ideen sprudeln, halten Sie keine Ideen zurück, sondern äußern Sie alles, was Ihnen in den Sinn kommt. Spinnen Sie die Ideen der anderen Teilnehmer weiter. Nach Ende der vorher vereinbarten Zeit werten Sie die einzelnen Beiträge aus. Um wirklich nichts zu verpassen, sollten Sie während der ganzen Sitzung einen Kassettenrecorder laufen lassen, damit Sie am Ende eine vollständige Liste haben.

Das 6-Farben-Denken

Bei dieser Methode werden bestimmten Sichtweisen Farben zugeteilt. Weiß steht für Neutralität, Daten und Fakten – keine eigenen Meinungen, sondern nur kalte Fakten und Logik.

Rot steht für Gefühl und Ahnungen. Meinungen und Gefühle können geäußert werden, ohne sie begründen zu müssen. Das Bauchgefühl ist entscheidend. Fakten und Logik spielen keine Rolle.

Schwarz steht für mögliche Fehler und Schattenseiten – die aber logisch begründet sind. Während weiß neutral ist, ist schwarz eher negativ.

Gelb steht für objektive positive Ideen und Gedanken, und ist quasi das Gegenteil von Schwarz. Es sollen alle machbaren, logisch begründbaren positiven Elemente gefunden werden. Es sollen keine neuen Ideen gefunden werden, sondern nur die »guten« Aspekte einer bestehenden Idee gefunden werden.

Grün steht für neue Ideen, Alternativen und frische Gedanken. Es geht darum, über die bisherigen Grenzen hinauszudenken, alles was zu neuen Ideen führt ist erlaubt - egal wie unglaublich die neuen Ideen sind. Jede Art von Kritik ist untersagt (dafür steht Schwarz)

Blau ist die Farbe der Kontrolle und Organisation. Ergebnisse werden zusammengefasst, geordnet und in Beziehungen gestellt.

Weisen Sie jedem Teilnehmer ein Farbe zu, und lassen Sie ihn zur Fragestellung im Namen seiner Farbe antworten. Jeder nimmt Reihum Stellung, dann werden die Farben neu verteilt. Nachdem jeder einmal jede Farbe hatte, geht es in Phase 2. In dieser Phase setzen alle einen bestimmten Hut auf, um einen bestimmten Aspekt näher zu beleuchten.

Die Walt-Disney-Strategie

Die Disney-Strategie ist der 6-Farben-Methode ähnlich. Auch sie zielt darauf ab, einzelne Aspekte nacheinander und getrennt zu beleuchten. Walt Disney hat 3 Rollen festgelegt: den Träumer, den Realisten und den Kritiker. Es ist äußerst wichtig, dass Sie diese Rollen nicht vermischen. Der Träumer sollte keinen Gedanken an die Umsetzbarkeit seines Traums verschwenden, der Realist sollte sich nicht mit möglicher Kritik belasten etc.

Der Träumer

Der Träumer ist die erste Rolle, die Sie einnehmen. Phantasieren Sie einfach darauf los. Es ist egal, ob die Ideen umsetzbar oder auch nur logisch sind. Träumen Sie ... es ist egal, ob sich die Ideen rentieren oder möglich sind ..., träumen Sie ..., alles ist möglich...

Der Realist

Die zweite Rolle ist der Realist. Betrachten Sie Ihren Traum und überlegen Sie, wie man ihn umsetzen kann. Nicht ob man ihn umsetzen kann, oder wie aufwendig oder rentabel er ist. Versuchen Sie Möglichkeiten zu finden, den Traum umzusetzen ohne ihn zu werten.

Der Kritiker

Die letzte Rolle ist der Kritiker. Er sucht nach Schwachpunkten in der Umsetzung des Traums. Seien Sie in der Rolle des Kritikers schonungslos und zerlegen Sie den Plan so gut Sie können. Suchen Sie nach Fehleinschätzungen und Illusionen. Hinterfragen Sie die finanzielle Machbarkeit.

Und nun geht es wieder von vorne los. Werden Sie wieder zum Träumer, und erweitern Sie Ihren Traum, um die vom Kritiker aufgeführten Punkte zu eliminieren. Wichtig ist, dass Sie den Traum nur erweitern und ihn nicht abschwächen. Wenn der Aufwand an Grafik zu hoch ist, dann träumen Sie sich eine Gruppe von Grafikern, die nichts anderes tun als neue Figuren und Animationen für Ihr Spiel zu erschaffen. Sie können diese Folge beliebig oft durchspielen. Mit der Zeit kommen dabei die drei Rollen ins Gleichgewicht. Normalerweise haben wir einen stärkeren Realisten oder Kritiker in uns – durch diese Methode wird der Träumer gestärkt, ohne dabei die anderen Rollen zu schwächen.

Das Pferd von hinten aufzäumen

Bei dieser Methode gehen Sie die Aufgabe von der anderen Seite an. Anstatt zu fragen, wie Sie das Spiel besser machen können, listen Sie Dinge auf, welche den Spielespass verringern. Anstatt Elemente aufzuzählen, die unbedingt im Spiel vorkommen sollen, suchen Sie nach allem, was tunlichst nicht benutzt werden sollte.

Diese Methode ist sehr gut geeignet, um eine Sitzung wiederzubeleben, wenn sie sich totgelaufen hat. Durch den Wechsel der Perspektive kommt ein frischer Wind in die Sache.

Der morphologische Kasten

Diese Methode erlaubt die systematische Ausarbeitung aller möglichen Lösungsrichtungen. Das Wort Morphologie kommt aus dem Griechischen und bedeutet »Lehre von den Gestalten, Formen und Strukturen«.

Diese Methode hat drei Arbeitsschritte:

- ✔ Zerlegen der komplexen Sachverhalte in einfache, abgrenzbare Komponenten
- ✔ Variation der Komponenten
- ✔ Zusammenfügen der einzelnen Teile zu einem neuen Ganzen

Nehmen wir an, Sie wollen das bestehende Inventarsystem verbessern. Im ersten Schritt zerlegen Sie das Problem: Was sind die Dinge, die alle Inventarsysteme gemeinsam haben? Worin unterscheiden sich die einzelnen Systeme? Auf welche Weise arbeitet das Inventarsystem mit den anderen Teilen des Spieles zusammen?

Liste der Parameter:

- ✔ Art der Repräsentation
- ✔ Begrenzung des Inventars
- ✔ Position des Inventars
- ✔ Sichtbarkeit

Nun werden diese Parameter einzeln betrachtet, und man versucht für jeden die maximale Anzahl an möglichen Ausprägungen zu finden.

Parameter	Ausprägungen
Repräsentation	Liste Gatter Kreismenü
Begrenzung	Größe Anzahl Gewicht
Position	Teilbereich Gesamter Schirm Zentral Am Rand
Sichtbarkeit	Ständig sichtbar und aktiv Bei Bedarf eingeblendet Sichtbar, aber inaktiv »unsichtbar«

Tabelle 2.1: Ausprägung der Parameter

Im letzten Schritt beginnen Sie dann, die einzelnen Ausprägungen zu kombinieren. Dies kann entweder intuitiv oder nach einer bestimmten Aufgabenstellung geschehen.

Osbournes Checkliste

Diese Methode eignet sich besonders, um Ideen für die Verbesserung eines bestehenden Systems zu finden.

Was ist ähnlich?	Gibt es ähnliche, vergleichbare Ideen? Worin unterscheiden sie sich? Was ist ähnlich? Welche Parallelen gibt es?
Andere Anwendungen?	Kann man es auch für andere Dinge verwenden? Gibt es weitere Anwendungsmöglichkeiten, wenn man Kleinigkeiten ändert?

Anpassen?	Was suggeriert es? Gibt es ähnliche Beispiele? Wie kann man es besser in das System einfügen?
Verändern?	Was passiert, wenn man das Aussehen ändert? Mit einer anderen Form? Andere Position? Veränderungen in der Akustik (Sprachausgabe, neue Sounds etc.).
Vergrößern?	Kann man andere Funktionen integrieren?
Verkleinern?	Welche Funktionen sind überflüssig? Kann man es optimieren? In einzelne Komponenten zerlegen?
Umformen?	Kann man Abläufe und Reihenfolge ändern? Kann man die einzelnen Komponenten anders anordnen? Ursache und Wirkung vertauschen?
Umkehren?	Was ist das genaue Gegenteil der jetzigen Situation? Wie kann man die Funktion umkehren? Das Innere nach außen kehren? Das untere nach oben?
Kombinieren?	Mit was kann man es kombinieren? Kann man verschiedene Funktion zusammenlegen?

Tabelle 2.2: Osbournes Checkliste

Mythen, Sagen und Märchen als Inspiration

Helden, Götter und ihre Schlachten haben die Menschen schon immer interessiert. Schlachten epischer Größe, der Kampf des Guten gegen das Böse, fantastische Gegner, geheimnisvolle Reisen... diese Geschichten waren den Menschen seit je her so wichtig, dass sie mündlich von Generation zu Generation weitergegeben wurden, bevor diese Aufgabe von Schriftrollen und Büchern übernommen wurde.

Darüber hinaus sind die meisten Mythen geradezu ideale Vorlagen für Computerspiele. Es liegt also nahe, sich von ihnen inspirieren zu lassen.

Odysseus

Odysseus ist der König von Ithaka. Er ersann die List des trojanischen Pferdes. Seine Rückkehr von Troja nach Ithaka wurde zu der berühmten Odyssee, die 10 Jahre andauerte.

Seine Abenteuer umfassten Begegnungen mit einäugigen Riesen, Zauberinnen, die Menschen in Schweine verwandeln können, Meerungeheuer und noch vieles mehr. Auch bietet die Irrfahrt viele Möglichkeiten neue Abenteuer einzufügen – man kann ja nie wissen, an welches Ufer einen die Winde führen.

Herakles

Herakles (oder Herkules) ist einer der bekanntesten Helden. Sohn des Göttervaters Zeus und der (sterblichen) Alkmene. Er ist mit enormer Stärke gesegnet (er erwürgte zwei Schlangen als er noch in der Wiege lag und besiegte als Jüngling einen Löwen). Seinen größten Widersacher hatte Herakles in Hera, Königin der Götter und Frau des Zeus. Sie war verständlicherweise nicht gerade erfreut über all die Affären ihres göttlichen Gemahls. Aus Eifersucht trieb sie Herakles in den Wahnsinn, und in dieser Zeit der geistigen Umnachtung tötete er seine Frau und seine Kinder.

Um Buße zu tun, legte ihm das Orakel des Apoll in Delphi zwölf Aufgaben auf:

- ✔ Den Löwen von Nemea zu töten. Kein Pfeil und kein Schwert konnte die Haut dieses Monsters durchdringen. Als Herakles erkannte, dass seine Waffen keinen Schaden anrichten konnten, nahm er seine Keule, und betäubte den Löwen mit einem mächtigen Schlag. Dann erwürgte er ihn mit bloßen Händen. Er versuchte den Löwen das Fell abzuziehen, doch da keine Waffe die Haut durchdringen konnte gelang es ihm zu Anfang nicht. Erst als er anstatt seines Schwertes die Krallen des Löwen benutzte hatte er Erfolg. Aus der Haut des Löwen machte er sich Umhang und Helm, der ihn vor den meisten Angriffen schützen.
- ✔ Die 9-köpfige Hydra zu besiegen. Dieses Monster hatte 9 Köpfe. 8 der Köpfe wuchsen sofort nach, wenn man sie abschlug (in einigen Legenden heißt es sogar, dass 2 Köpfe für jeden abgeschlagenen Kopf nachwuchsen). Der 9te Kopf war unsterblich. Der Atem der Hydra war tödlich für Mensch und Tier. Herakles besiegte das Monster, indem er die Köpfe abschlug und dann den Halsstumpf mit einer Fakkel kauterisierte, sodass der Kopf nicht nachwachsen konnte. Den

unsterblichen 9ten Kopf begrub er in der Erde und rollte einen großen Stein darüber.

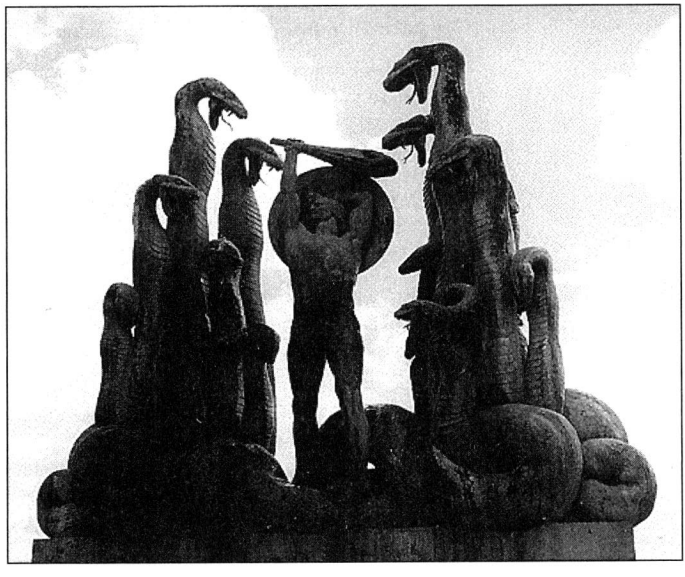


Abbildung 2.1: Herakles gegen die Hydra

- ✔ Die kerynitische Hirschkuh zu fangen. Herakles jagte die Hirschkuh für einige Monate bevor er in der Lage war sie zu fangen.
- ✔ Den erymantischen Eber zu besiegen. Auch dies war kein Problem für Herakles.
- ✔ Den Stall des Augeias an nur einem Tag auszumisten. Klingt wie eine leichte Aufgabe, jedoch beherbergte der Stall mehrere Rinderherden und war seit Jahren nicht mehr gereinigt worden. Herakles schaffte es nur, weil er einen nahen Fluss umlenkte und damit den Stall ausspülte.
- ✔ Die stymhalischen Vögel vertreiben. Diese Fleisch fressenden Raubvögel konnten nur mit Hilfe einer Rassel vertrieben werden, die Herakles von der Göttin Athene erhielt.

- ✔ Den Stier des Minos auf Kreta bändigen. Wieder eine eher leichte Aufgabe und Herakles konnte sie mit relativ wenig Aufwand bewältigen.
- ✔ Die Menschen fressenden Stuten des Diomedes fangen.
- ✔ Den Gürtel der Hippolyte zu beschaffen. Hippolyte ist die Königin der Amazonen und der Gürtel ist ein Zeichen ihrer Macht. Die Amazonen sind ein Volk von Kriegerinnen, und Hippolyte war eine Bewunderin von Herakles. Aus diesem Grund willigte Sie ein, ihm den Gürtel zu geben. Hera war davon nicht sehr erfreut und sorgte für einen Aufruhr und hetzte die Amazonen auf Herakles. In dem darauf folgenden Kampf wurde Hippolyte getötet, Herakles nahm ihr den Gürtel ab und entkam der tosenden Meute der Kriegerinnen.
- ✔ Die Rinder des geflügelten Giganten Geryones zu beschaffen. Das Problem hierbei bestand daran, dass dieser die Rinder nicht freiwillig abgeben wollte und sie von dem zweiköpfigen Hund Orthrus bewacht wurden.
- ✔ Die goldenen Äpfel der Hesperiden zu stehlen. Herakles löste diese Aufgabe, indem er Atlas überredete, die Äpfel für ihn zu besorgen. Atlas ist der Titan, dessen Aufgabe es ist, den Himmel zu tragen. Herakles bot ihm an, diese Last für eine Zeit zu übernehmen, wenn Atlas die Äpfel der Hesperiden holen würde. Atlas willigte ein, wollte dann aber die Last nicht wieder übernehmen. Herakles überredete ihn nur noch einmal kurz die Last zu tragen, damit er sich ein Kissen auf die Schultern legen könne. Sobald Atlas jedoch wieder den Himmel trug machte sich Herakles mit den Äpfeln auf und davon.
- ✔ Den Höllenhund Zerberus and die Oberwelt zu bringen. Zerberus ist der dreiköpfige Hund des Hades, der die Aufgabe hat, das Tor zu Unterwelt zu bewachen.

Perseus

Perseus war der Legende nach der Sohn des Göttervaters Zeus mit der Königstochter Danae. Perseus erhielt die Aufgabe, die Gorgon Medusa zu erschlagen. Die Medusa war eine Frau mit Schlangen als Haaren, deren Anblick jeden in Stein verwandelte.

Um sein Ziel zu erreichen, holte er sich Rat bei den Graeae, drei alten Frauen mit mythischen Fähigkeiten. Mit ihrer Hilfe bekam er drei magische Gegenstände: Ein Paar geflügelter Sandalen, die es ihrem Träger er-

lauben zu fliegen. Einen Sack, um das Haupt der Medusa aufnehmen zu können, und die Kappe des Hades, die ihren Träger unsichtbar macht. Mit diesen Gegenständen gewappnet näherte er sich unsichtbar der Medusa, wobei er seinen Schild als Spiegel benutzte, um sie nicht direkt ansehen zu müssen.

Auf seinem Rückweg flog er an Atlas vorbei, und nach einem Streit verwandelte Perseus ihn in Stein, in dem er ihn den abgeschlagenen Kopf der Medusa zeigte.

Dann kam er an einem Felsen vorbei, an dem die hübsche Jungfrau Andromeda als Opfer für ein Seeungeheuer angeschmiedet war. Er überredete ihren Vater, ihm Andromeda zur Frau zu geben, wenn es ihm gelingen sollte das Monster zu besiegen. Also kämpfte er mit dem Monster und besiegte es in einem anstrengenden Kampf.

Nach seiner Heimkehr übergab er die Sandalen, Beutel und Kappe an Hermes, den Götterboten. Den Kopf der Medusa bekam die Göttin Athene.

Weitere griechische Helden

Die griechischen Sagen sind voll von Helden: Theseus, Jason und seine Argonauten, Achilles und Ajax. Auch gibt es viele Personen, die in den großen Geschichten nur eine Nebenrolle einnehmen, aber hervorragende Protagonisten für Computerspiele abgeben würden. Da wäre zum Beispiel Autolycus, Sohn des Hermes. Dieser war ein geschickter Dieb und exzellenter Ringkämpfer. Außer seiner Teilnahme an der Suche nach dem Goldenen Vlies ist nicht viel über ihn bekannt. Aber seine Fähigkeiten und seine göttliche Herkunft könnten ihn ideal für ein auf Taktik und Schleichen ausgelegtes Spiel machen.

Nordische Sagen als Inspiration

Odin, der Göttervater

Odin ist der Gott des Krieges und des Todes, aber auch der Dichtkunst und Weisheit. Seine beiden Raben versorgen ihn mit allen Informationen, die er braucht. Seine Waffe ist der Speer Gungir und er reitet auf einem Ross mit sechs Beinen und wird von 2 Wölfen begleitet.

Er hat nur ein Auge, da er sein zweites für einen Schluck aus der Quelle des Wissens eintauschte. Er durchstreift gerne die Welten und hat aus diesem Grunde auch den Beinamen »Der Wanderer«.

Abbildung 2.2: Odin

Er ist nicht nur ein mächtiger Krieger, sondern auch begabt in den magischen Künsten.

Thor, Gott des Donners

Thor ist ein Gott der nordischen Mythologie. Er ist der Sohn des Odin und wird normalerweise als kräftiger Mann mit rotem Bart und strahlenden Augen dargestellt. Er trägt den Gürtel Megingjard, der seine bereits beträchtliche Körperkraft noch mal verdoppelt. Seine bekannteste Waffe ist der Zauberhammer Mjöllnir, der es ihm erlaubt Blitze zu schleudern. Wirft er den Hammer, so kommt dieser immer wieder zu ihm zurück. An seiner rechten Hand trägt er einen eisernen Handschuh, mit dem er seinen Hammer fängt.

Heimdall, Gott des Lichts

Heimdall ist der Wächter der Brücke Bifrost, die Asgard, die Welt der Götter, mit den anderen Welten verbindet. Seine Aufgabe ist es zu verhindern, dass die Riesen in Asgard eindringen können.

Freyr, Gott der Ernte

Freyr ist nicht nur der Gott der Ernte, sondern außerdem auch ein tapferer Krieger und der Herrscher der Elfen. Er besitzt das Schiff Skidbladnir, das immer direkt auf das Ziel zusteuert und auf sein Kommando auf Miniaturgröße zusammenschrumpft. Er kann es dann einfach in seine Tasche stecken. Er besitzt außerdem noch ein magisches Schwert, welches auf Befehl selbstständig kämpft.

Sie können Sagen entweder direkt umsetzen, oder Ihr Spiel einfach nur auf diesen Sagen aufbauen. Verlegen Sie den Handlungsort doch mal in den Cyberspace oder eine ferne Zukunft. Erweitern Sie die Legenden, geben Sie Ihren Helden neue Aufgaben, die sie zu erfüllen haben. Lassen Sie Herakles mit Thor in eine Schlacht ziehen. Die Möglichkeiten sind beinahe endlos.

Natürlich können Sie auch komplett neue Geschichten mit neuen Helden erschaffen.

3 Geschichten schreiben

Was macht eine gute Geschichte aus? Wie erschafft man in sich schlüssige Welten? Diese beiden Fragen werden in diesem Kapitel beantwortet.

Zwar braucht nicht jedes Spiel eine packende Hintergrundgeschichte, schaden kann sie jedoch niemals. Und gerade bei Rollenspielen kann die Hintergrundgeschichte den Spieler in den Bann ziehen.

In »Der Heros in den tausend Gestalten« beschreibt Campbell den prinzipiellen Aufbau der Heldensagen und Mythen. Diesen Aufbau kann man als grobe Richtlinie für eine Geschichte benutzen. Wenn Sie diesen Aufbau kennen, werden Sie ihn auch in vielen anderen Geschichten erkennen können. Seien es nun die Abenteuer des Herakles, Odysseus oder die von Luke Skywalker – sie alle haben den gleichen Aufbau.

Da der generelle Aufbau vorgegeben ist, können Sie die gesamte Story viel schneller planen. Zögern Sie nicht einzelne Aspekte zu ändern oder auch mehrere komplett verschiedene Entwürfe zu schreiben.

Die Heldenreise

Die Heldenreise beschreibt die einzelnen Etappen der Entwicklung der Hauptperson. Die Liste, die ich Ihnen hier präsentiere, ist nicht in Stein gemeißelt. Lassen Sie Punkte weg, die Ihnen nicht gefallen, fassen Sie andere zusammen. Es gibt auch keinen Grund das Spiel am Anfang der Heldenreise beginnen zu lassen. Ebenso gut kann der erste Teil im Vorspann erzählt werden.

Die normale Welt

Zu Beginn erleben wir den späteren Helden in seiner normalen Umgebung. Er ist noch ein Mensch wie jeder andere auch. Nie im Traum würde er daran denken, dass aus ihm einmal ein großer Held werden würde. Er geht seiner normalen Beschäftigung nach: Kinder spielen, ein Schmied würde ein paar Hufeisen machen und der Bauer nach seinen Tieren sehen.

Ein Feuchtigkeitsfarmer auf einem Wüstenplanet könnte zum Beispiel gerade ein paar neue Roboter kaufen, eine junge Teenagerin einkaufen gehen und ein Kung-Fu-Mönch durch das Land ziehen.

Der Ruf des Abenteuers

An diesem Punkt im Leben des Helden wird klar, dass sich etwas ändert wird. Vielleicht merkt der Held es selbst nicht einmal (oder will es nicht wahrhaben). In »Krieg der Sterne: Eine neue Hoffnung« ist es Obiwan Kenobi, der Luke seine spätere Bestimmung zeigt. Er übergibt ihm das Lichtschwert seines Vaters – damit beginnt für ihn ein neuer Abschnitt seines Lebens.

In einem Kung-Fu-Film ist dies die Stelle, an der der Held zum ersten Mal mit den Bösewichten in Kontakt kommt. Für den Helden selbst ist es meist noch ein unbedeutender Zwischenfall, dem Betrachter / Leser wird jedoch klar, dass sich hier die Wende vollzieht.

Die Weigerung

Das Abenteuer ruft, aber der Held stellt sich taub. Nur in den wenigsten Fällen werden sich Leute bereitwillig in ein Abenteuer stürzen. Meist gibt es eine Arbeit, die es erst noch zu vollenden gilt, Menschen, um die man sich kümmern muss, oder einfach nur Bedenken, ob es richtig ist, ins Abenteuer zu ziehen.

Luke muss noch auf der Farm helfen, der Kung-Fu-Kämpfer hat geschworen sich nur zu verteidigen und will deswegen nicht in den Kampf ziehen. Und die Teenagerin glaubt vielleicht einfach nicht, dass sie auserwählt ist gegen das Böse zu kämpfen.

Überschreiten der ersten Grenze

Der Held wird sich bewusst, dass er keine andere Wahl hat als sich dem Abenteuer zu stellen. Entweder werden die Bindungen zur normalen Welt durchtrennt, der Held muss einen schweren Verlust erdulden oder eine geheimnisvolle Macht klärt ihn über seine wahre Bestimmung auf.

Luke Skywalker steht vor den Überresten der vom Imperium zerstörten Farm und beschließt, sich Obiwan anzuschließen.

Ein Monster/Vampir/Werwolf greift eine Freundin der Heldin an, der Mönch hat ein klärendes Gespräch mit seinem Abt.

Die Prüfungen

Dem Helden werden eine Reihe von Prüfungen, Herausforderungen und Aufgaben gestellt. Nicht immer gelingt es ihm Sie alle zu lösen, doch er zeigt guten Willen. In einem Spiel ist dies der Zeitpunkt, zu dem die ersten richtigen Kämpfe auf den Helden zukommen. Ein Kämpfer könnte sich eventuell einen neuen Lehrmeister suchen und von diesem hart trainiert werden.

Treffen mit der Göttin/Gefährten

Nach all den körperlichen und geistigen Herausforderungen wartet nun eine emotionale Herausforderung auf den Helden. So wichtig diese Phase für die Tiefe des Charakters notwendig ist, so schwer ist es auch sie auf eine nicht schwülstig wirkende Weise in einem Spiel unterzubringen.

In den meisten Fällen wird es hier wohl auf eine Zwischensequenz mit mehr oder weniger romantischer Prosa hinauslaufen – obwohl es sicher interessant wäre dem Spieler an dieser Stelle einige Freiheiten zu lassen.

Nun wird es auch Zeit, dass der Held Mitstreiter findet, die an seiner Seite für das gleiche (oder zumindest ein ähnliches) Ziel zu kämpfen bereit sind.

Weitere Prüfungen und Verlust des Mentors

Die Gefährten sind nun auf einer gefährlichen Reise, meist durch eine Art von Labyrinth oder Kerker. Der Held und seine Gefährten müssen sich den ersten schwierigen Prüfungen stellen. Zumeist überlebt der Mentor, der weise Führer der Gruppe diesen Schritt nicht (oder gilt als verschollen). Doch die Gruppe findet neue Fertigkeiten oder tieferes Wissen, das ihnen bei der eigentlichen Aufgabe deutlich von Vorteil sein wird.

Der Sieg

Der Held stellt sich der eigentlichen Aufgabe. All die vorangegangen Schritte dienen nur dazu diesen Teil vorzubereiten. Dies ist die finale

Schlacht, der Höhepunkt der Geschichte. Auf irgendeine Weise kehrt auch der verlorene Freund wieder zurück. Entweder indem er wirklich in Erscheinung tritt, oder dem Helden nur im Geiste oder als Erinnerung beisteht. In manchen Fällen ist es auch einfach nur die Erinnerung an eine Lehre des Meisters, die den Unterschied zwischen Niederlage und Sieg ausmacht.

Die Rückkehr

Dies ist das abschließende Kapitel. Hier sollten alle losen Fäden zusammenlaufen, und die Motive von beiden Seiten offengelegt werden.

Die Heldenreise angewendet

Nun geht es darum, dieses theoretische Wissen umzusetzen. Gehen wir davon aus, dass bisher diese Punkte feststehen:

- ✔ Das Spiel soll einen Schwerpunkt auf Kämpfen und Magie haben.
- ✔ Der Held soll in der Lage Kreaturen zu beschwören.
- ✔ Der Held soll gut mit dem Schwert umgehen können.

Die letzten beiden Punkte scheinen sich etwas zu beißen (Gut kämpfende Magier? Wo gibt's denn so was!), als brauchen wir eine gute Erklärung.

Der Held des Spieles, Knurd Eisenfaust, stammt aus einer Familie von Kämpfern. Seit er stark genug war das Schwert seines Vaters zu bewegen, trainiert er seine Angriffe. Sein großes Ziel ist es bei dem Schwertmeister des Nachbardorfes in die Lehre gehen zu können.

In dem Jahr, in dem er seinen sechzehnten Geburtstag feiert, geht Knurd in die Stadt, um sich von den Dorfältesten seine Bestimmung als Kämpfer bestätigen zu lassen. An dem Tag der Auswahl versammelt sich das gesamte Dorf, und alle arbeitsfähigen Jugendlichen werden von den lokalen Handwerkern, Magiern, Kämpfern und Priestern auf eine Eignung für den jeweiligen Beruf getestet.

Der Beschützer der Stadt, ein inzwischen etwas ergrauter Kämpfer und Monsterjäger, kennt Knurd gut und begrüßt ihn mit freudigen Worten. Der Magier bereitete inzwischen seine Kristallkugel vor, um die Jugendlichen auf ihre magische Fähigkeiten zu testen. Nach kurzer Zeit sind alle soweit, und das Ritual kann beginnen.

Jeder der Jugendlichen tritt vor, legt seine Hand auf die Kristallkugel und sagt seinen Namen. Bei denen ohne magische Begabung färbt sich die Kugel blau, ist eine Begabung für göttlichen Zauber vorhanden, färbt sie sich grün und bei denen mit arkanen Talenten golden.

Bei den Jugendlichen vor Knurd gibt es keine Überraschungen. Zwei neue Priesteranwärter werden gefunden und die beiden anderen haben kein magisches Talent und gehen zu den Handwerkern.

Als Knurd seine Hand auf die Kugel legt, passiert etwas Seltsames – anstatt sich blau zu färben, wie es alle bei dem kämpferischen Knurd erwartet haben beginnt die Kugel zu vibrieren und alle Farben des Regenbogens in schneller Folge zu durchlaufen. Erschreckt zieht Knurd die Hand zurück. Ein Blick in die Runde zeigt nur ratlose und erstaunte Gesichter.

Endlich spricht der Magier: »Ich habe ein solches Verhalten der Kugel nur einmal gesehen. Es scheint, dass Ihr ein äußerst seltenes Talent habt, werter Knurd: Ihr seid ein Beschwörer!«

Diese Einleitung bietet einen guten Einstieg für die Heldenreise. Knurd will kein Magier und schon gar kein Beschwörer sein (er wusste bis dahin nicht einmal, was ein Beschwörer ist). Dies bietet einen guten Ausgangpunkt für die Weigerung und das Treffen mit seinem Mentor.

Möglicherweise soll er sich auf die Reise in eine entlegene Stadt begeben, um dort einen der wenigen anderen Beschwörer zu treffen. Er stimmt dieser Reise nur zu, weil sein guter Freund Morv einen ähnlichen Weg hat und nicht allein reisen will. Insgeheim hat er vor, den Beschwörer nicht zu suchen, sondern Morv zu begleiten.

Auf dem Weg dahin kann er seine Fähigkeiten als Kämpfer beweisen. In einem der heftigeren Kämpfe wird Morv schwer getroffen, und der Kampf scheint verloren. Plötzlich tauchen wie aus dem Nichts zwei Wölfe auf und vertreiben die Angreifer. Als Knurd sich umsieht entdeckt er einen alten Mann in einem weiten Umhang. Mit einem Fingerschnippen lässt er die Wölfe verschwinden und kümmert sich dann um den verwundeten Morv.

In dem nun folgenden Gespräch wird schnell klar, dass Knurd seinen neuen Lehrmeister gefunden hat. Dankbar für die Hilfe und auch beeindruckt von den Fähigkeiten des Mannes schließt er sich ihm an und wird sein Lehrling.

In seiner Lehrzeit muss er viele kleinere Aufgaben bestehen, in Höhlen nach magischen Kräutern und seltsamen Artefakten suchen. Und er lernt die Kunst des Beschwörens.

Eines Tages taucht ein Bote auf und wünscht Knurds Mentor zu sehen. Nach kurzer Zeit wird klar, dass Not am Manne ist, und Knurd und der alte Beschwörer ziehen los das magische Artefakt »Drachenauge« aus der Höhle der Dunkelheit zu retten. Finstere Anhänger schwarzer Magie wollen das Drachenauge an sich reißen, um seine Macht zur Beherrschung des Landes zu nutzen. Am Eingang der Höhle kommt es zu einem Aufeinandertreffen zwischen den Beschwörern und einem mächtigen Geist, den die schwarzen Magier als Bewacher der Höhle gerufen haben.

Als der Geist Knurd mit einem vernichtenden Zauber angreift, springt sein Mentor vor ihn und nimmt die vernichtende Energie in sich auf. Mit seiner letzten Kraft bannt er den Geist und sinkt dann tot zu Boden.

Nun obliegt es Knurd allein, das Drachenauge zu finden und damit das Land vor dem Untergang zu bewahren...

In der Höhle wird Knurd viele Kämpfe zu bestehen haben, die Fallen der Schwarzmagier als auch die zum Schutz des Drachenauges geschaffenen Rätsel lösen müssen.

Glücklicherweise kann er in Schatztruhen neue Waffen finden, damit er für den Endkampf gewappnet ist. Nach dem Finalkampf erhält er das Drachenauge und kehrt als Held in sein Heimatdorf zurück.

Weitere Planung

Diese Geschichte ist sicher kein literarisches Meisterwerk, sie ist aber auch nicht deutlich schlechter als die der meisten Rollenspiele. Und mit etwas mehr Arbeit und ein paar Änderungen könnte dies sogar ein sehr gutes Rollenspiel ergeben. Nutzen Sie doch die Kreativitätstechniken, die in den vorangegangen Kapitel erläutert wurden, um der Geschichte ein paar interessante Wendungen zu geben.

Welten erschaffen

Eine einfache Geschichte wie die oben skizzierte kann in beinahe jeder Fantasy-Welt ablaufen. Aber was, wenn die Welt zu einem Teil des Abenteuers werden soll? Wie vermeidet man logische Fehler in einer Welt?

Nehmen wir doch mal die »normale« Vampirgeschichte als Ausgangspunkt. Der alte Graf lebt (oder besser: existiert) auf seinem unheimlichen

Schloss. Die Dorfbewohner wissen alle, dass es sich bei ihm um einen Vampir handelt und begegnen ihm mit Furcht und Zorn.

Da Vampire bekanntlich Blut brauchen ist anzunehmen, dass der Graf des Nachts gerne mal an den Dorfbewohnern nascht. Worauf diese ihren Knoblauchvorrat deutlich aufstocken und immer ein Kruzifix bereit haben.

Dies würde eigentlich bedeuten, dass der Graf immer weitere Ausflüge unternehmen muss, um noch seinen täglichen Tropfen Blut zu bekommen. Aber früher oder später wird dies ein zeitliches Problem. Er muss ja vor Sonnenaufgang wieder in seinem Sarg liegen.

Diese Ausflüge würden natürlich auch in der Umgebung das »Vampirbewusstsein« stärken, und irgendwann sollte die Existenz der Vampire dann kein Geheimnis mehr sein. Auch ist die Chance sehr hoch, dass sich ein bewaffneter Mob sammelt, um dem Grafen bei Tage zu Leibe zu rücken.

Und doch scheint es so, als ob die Dorfbewohner es immer irgendwie schaffen sich nachts vom Grafen beißen zu lassen... trotzdem sinkt die Einwohnerzahl der Stadt nicht. Und der Mob kommt meist nur dann zum Einsatz, wenn der Held gerade den Vampir besiegt hat.

Wird eine Stadt überraschend von Vampiren heimgesucht, dann schaffen es alle Bewohner die offensichtlichen Anzeichen wie offene Gräber, herumlaufende Tote und so weiter bis zum letzten Moment zu ignorieren.

Nur der Held, meist ein durch das Lesen von Comicbüchern zu einem Vampirexperten geworden, erkennt die drohende Gefahr und tritt mit einer mit Weihwasser gefüllten Spritzpistole und den obligatorischen Pflöcken an, das Böse zu besiegen.

Seien wir doch mal ehrlich: Diese Geschichten haben doch mehr Lücken als ein durchschnittliches Netz.

Nehmen wir mal an der erste Vampir ist so um 30 nach Christie Geburt entstanden (wie eine beliebte Sage behauptet). Gehen wir auch davon aus, dass Vampire nicht altern, und nur durch Sonnenlicht, Enthauptung oder einen Pflock getötet werden können. Heilige Symbole schrecken sie ab und aufgrund ihrer guten Nase haben sie eine Abneigung gegen Knoblauch.

Jetzt muss noch geklärt werden, ob es reicht von einem Vampir gebissen zu werden um selbst zum Vampir zu werden, oder ob man das Blut eines Vampirs trinken muss, um verwandelt zu werden.

Je nachdem welche Variante gewählt wird, ändert sich die Welt dementsprechend. Wenn ein Vampir einmal pro Woche einen Menschen beißt, dann haben wir nach einem Monat 8 Vampire, nach 2 Monaten bereits 256 Vampire und nach 8 Monaten sind es bereits über 4 Millionen Vampire.

Woche	Vampire
1	1
2	2
3	4
4	8
...	
8	256
16	65.536
32	4.294.967.296

Tabelle 3.1: Vampirwachstum

Bei dieser Menge an Vampiren und ihrer enormen Widerstandskraft hätten normale Menschen wohl kaum eine Chance zu überleben. Städte könnten in einer Nacht überrannt werden. Und Menschen würden wohl wie Vieh gehalten, um den Vampiren als Nahrung zu dienen.

Gehen wir davon aus, dass die Menschen sich der Gefahr bewusst werden und dann einen Feldzug gegen die Vampire führen. Gehen wir auch davon aus, dass die Menschen es irgendwie schaffen zu gewinnen. Dann würden die verbleibenden Vampire sich wohl zu Geheimbünden zusammenschließen und versuchen unerkannt zu bleiben. Das Gleiche werden wohl auch die Menschen machen und geheime Kulte und Orden der Vampirjäger schaffen. Ein einzelner Vampir sollte dann also versuchen nicht erkannt zu werden, um nicht die Aufmerksamkeit der Jäger auf sich zu lenken. Der Graf auf seinem Schloss wird dann wohl versuchen nicht zu viel Schrecken zu verbreiten.

Wenn Vampire jedoch wirklich dunkle Geschöpfe der Nacht sind, dann ist die Chance groß, dass sie sich nur solange an diese geheime Abmachung halten, wie sie einen Vorteil darin sehen. Wenn sie das Gefühl haben, das Gleichgewicht der Macht zu ihrem Vorteil verschieben zu können, dann werden sie das wohl auch versuchen.

Wenn die Jäger das merken, werden sie natürlich Gegenmaßnahmen ergreifen. Es ist also anzunehmen, dass alle paar Generationen ein Krieg von Menschen gegen Vampire stattfindet.

Mit der Zeit werden sich auch Menschen mit Vampiren verbünden (um mehr Macht zu erhalten) und Vampire versuchen, ein möglichst menschliches Leben zu führen.

Ein Spiel in einem solchen Setting hätte dann wohl sehr viel Ähnlichkeit mit einem Agenten- oder Verschwörungsthriller (es sei denn, es spielt in der Zeit eines Vampirkriegs).

Es könnte natürlich auch sein, dass die Vampire es nicht schaffen sich zu verbreiten, weil sie immer sehr schnell von wütenden Mobs gelyncht werden. Aber je nach Ursprungstheorie könnten alle paar Jahrhunderte neue Vampire entstehen, die dann wieder versuchen mit der neuen Situation zurecht zu kommen. In diesem Fall wären die Anti-Vampir-Organisationen deutlich kleiner und weniger einflussreich.

Will man Vampire in der heutigen Zeit ansiedeln, dann ist es ratsam, eine Methode zu finden, die damals ausgerotteten Vampire in der heutigen Zeit wiederzubeleben.

Man könnte einen »verrückten Wissenschaftler« als Story Element benutzen. Dieser könnte versuchen, das Elixier des ewigen Lebens zu erschaffen. Auf seiner Suche nach einer Möglichkeit findet er das Grabmal eines alten Vampirs und benutzt dessen Überreste, um den »Vampir-Virus« zu isolieren.

Nun könnte entweder ein furchtbarer Unfall passieren und dieser Virus freigesetzt werden, oder der Wissenschaftler verschätzt sich etwas, und das Elixier hat einige Nebenwirkungen. Das Ergebnis wäre eine Vampirähnliche Kreatur die aber (aufgrund der Änderungen des Wissenschaftlers) andere Schwächen (und eventuell auch Stärken) als ein klassischer Vampir hat. In einem solchen Setting würde der Spieler quasi die Geburt des Vampirs miterleben. Die Auswirkungen auf die Spielwelt wären also eher gering.

Dieses einfache Beispiel zeigt deutlich, wie selbst das Hinzufügen einer einzigen fantastischen Spezies in unsere Welt enorme Auswirkungen hat.

Magie

Welche Auswirkungen hätte funktionierende Magie auf die Gesellschaft? Je nach Stärke der Magie (was kann man mit ihr bewirken) und Anzahl der Magier wären die Auswirkungen wohl sehr unterschiedlich. Wenn jeder, oder fast jeder Magie wirken kann, dann würde sie sehr schnell ein normaler Teil des Lebens werden. Es gäbe Magie Unterricht an Schulen, und viele Berufe hätten sich wohl gar nicht entwickeln können. Wer braucht schon einem Arzt wenn der freundliche Zauberer aus der Nachbarschaft einfach einen Heilzauber wirken kann?

Wenn Magie nicht so alltäglich ist, dann würden Zauberer eine andere Rolle annehmen. Die Menschen hätten Respekt vor Ihnen – oder einfach nur Angst vor Ihrer Macht. In einer solchen Umgebung könnte sich auch Technologie entwickeln, da das normale Volk sich keine Magier leisten kann.

Würden die Magier die Entwicklung der Technik überhaupt zulassen oder sich durch sie bedroht fühlen? Gefährliche Waffen wie Musketen und Bomben würden von den Zauberern wohl auf jeden Fall misstrauisch überwacht werden.

Könige und Adlige werden sich wohl sicher einen oder mehrere Magier leisten, und in jeder größeren Stadt gibt es sicherlich einen oder mehrere Zauberer für die Belange der Allgemeinheit.

Je nach Anzahl und Macht der Magier wird die Bedeutung der Technik und Naturwissenschaften immer geringer. Je größer die Macht der Magier, umso misstrauischer werden die normalen Menschen. Besonders Zusammenschlüsse von Magiern dürften den Mächtigen der Welt wohl verdächtig vorkommen und aus diesem Grund sehr genau beobachtet werden.

Priester und göttliche Magie

Normalerweise sorgen in einem Rollenspiel Magier für die Feuerbälle und Priester heilen die Verwundeten. Aber warum soll dies so sein? Wel-

che Macht, welche Art von Magie steht den Magiern zur Verfügung? Worin unterscheidet sich die Magie der Priester von denen der Magier?

Um diese Fragen zu beantworten, ist es wichtig zu klären, was Götter im Sinne des Spieles eigentlich sind. Menschen schreiben Aspekte der Natur den Göttern zu, seien es Donner und Blitz, Fruchtbarkeit, Schönheit, Schutz vor Krankheiten, Krieg und Tod etc. Auch suchen Menschen Schutz bei mächtigeren Wesen, aus diesem Grund gibt es Schutzgötter für beinahe jede Berufsgruppe. Manchmal werden Götter auch mit bestimmten Regionen oder Klimazonen in Verbindung gebracht (Gott des Waldes, Gott des Meeres etc.). In der klassischen Literatur sind die Götter jedoch auch nicht allmächtig. Sie müssen sich ebenso an die »Spielregeln« halten wie die Sterblichen.

Wenn Götter die gleiche Art von Magie benutzen wie Menschen, und Priester in der Lage sind durch ihre Verbindung zu ihrem Gott Zauber zu wirken, dann sollten sich die Zaubersprüche von Magiern und Priestern nicht so sehr unterscheiden.

Um nun die Wahl zwischen Priester und Magier interessanter zu machen, kann man die Stärke und den Zugang zu den einzelnen Sprüchen variieren. Ein Priester kann nur die Zaubersprüche nutzen, die sein Gott ihm gewährt. So kann ein Priester des Donnergottes Blitze schleudern und das Wetter kontrollieren, ein Priester des Waldgottes mit Tieren und Bäumen sprechen und ein Priester des Kriegsgottes seine Kampfkraft verstärken.

Wenn Sie für Ihr Spiel eigene Religionen erschaffen möchten, dann stellen Sie sich folgende Fragen:

- ✔ Wie steht die Glaubensrichtung zu Ideen von außen oder neuen Ideen im Allgemeinen?
- ✔ Beten die Menschen nur einen einzelnen Gott an oder mehrere?
- ✔ Wie steht die Religion zu anderen Religionen?
- ✔ Gibt es Vermittler wie Priester, Äbte, Mönche und Pfarrer, die zwischen dem normalen Mann auf der Straße und seiner Gottheit stehen, oder wenden sich die Anhänger direkt an ihren Gott?
- ✔ Macht die Religion Unterschiede nach Geschlecht, Rasse, sozialem Status oder politischen Überzeugungen?
- ✔ Welche Art von Ritualen, heiligen Gegenständen und Feiertagen hat die Religion?

- ✔ Gibt es ein Zeichen für die Glaubensrichtung? Heilige Symbole? Tätowierungen oder anderer Körperschmuck? Welche Art von Kleidung tragen die Priester des Glaubens?

Wenn Sie Antworten auf diese Fragen haben und wissen, welche Stärken und Schwächen die jeweiligen Götter haben, dann sollten Sie auch in der Lage sein, die verfügbaren Zaubersprüche den jeweiligen Priestern zuzuordnen.

Drachen

Wenn es in einer Welt Magie gibt, dann gibt es meist auch Drachen. Und wie wir spätestens seit »Jurassic Park« wissen, haben große Raubtiere eine enorme Auswirkung auf das Ökosystem. Wenn es nun einen solchen großen Räuber geben würde, der keinen natürlichen Feind (außer sich selbst) hat, dann sähe unsere Welt anders aus.

Menschen würden wohl nur in nicht brennbaren Unterkünften leben und alle Tiere würden bei dem Geräusch großer, lederner Schwingen sehr unruhig werden. Wenn die Drachen all die fantastischen magischen Fähigkeiten und auch die große Intelligenz haben, die ihnen in manchen Geschichten nachgesagt wird, dann ist es sehr wahrscheinlich anzunehmen, dass einige Drachen wohl auch Gefallen daran finden würden, über die Menschen zu herrschen.

Und Drachen sind ja nicht die einzigen fantastischen Tiere. Ein Fantasy-Rollenspiel wimmelt ja geradezu von Ungeheuern und grausigen Tieren. All diese Tiere müssen in die normale Nahrungskette eingegliedert werden, um eine realistische Welt zu erzeugen.

Zusammenfassung

Eine Welt muss nicht immer zu 100% durchdacht sein. Riesige Mengen an Monstern, die die Wälder durchstreifen, hätten sicherlich einige schwerwiegende Auswirkungen auf die Bevölkerung. In den normalen Rollenspielen kann man kaum mal 10 Minuten außerhalb eines Dorfes herum laufen, ohne in ein Monster zu rennen. Bei dieser Monsterdichte und den kämpferischen Fähigkeiten des typischen Rollenspiel-Dorfbewohners wäre das Dorf sehr schnell menschenleer.

Auch ist es sehr unwahrscheinlich, dass noch kein anderer den Höhleneingang gefunden und den Schatz des Affenkönigs vor uns geborgen hat.

Trotzdem ist das Spiel mit einer hohen Monsterdichte und Höhlen voller Schätze deutlich ansprechender. Lassen Sie nicht zu, dass sich der Realismus zwischen Sie und ein interessantes Spiel stellt.

Lassen Sie den Spieler spielen

Ein Spiel ist ein Spiel, weil der Spieler damit spielt. Klingt doch recht einleuchtend, oder? Doch manchmal kann man das Gefühl bekommen, dass einige Rollenspiel-Entwickler lieber Filme drehen oder Bücher schreiben würden. Wenn Sie eine geniale Geschichte haben, die Sie gerne in der Form eines Rollenspiels erzählen wollen, dann stellen Sie sicher, dass sich die Story beim Spielen entfaltet. Wenn Sie regelmäßig Zwischensequenzen brauchen, um die Story voranzutreiben, dann stimmt etwas nicht. Denn in diesem Fall hätten Sie zwei verschiedene Ebenen: Das Spiel und die Story. Dadurch würde die Geschichte etwas, das in Spielpausen passiert. Oder noch schlimmer: Das Spiel würde zu den kurzen Sequenzen zwischen den Erzählungen werden.

Stellen Sie sich vor, dass ein Barde in einem Kinofilm von einer großen Schlacht erzählt. Nun wird normalerweise während der Erzählung die Bilder des Kampfes eingeblendet oder es kann sogar sein, dass der Sprecher in den Hintergrund tritt und die Bilder die Geschichte erzählen.

Haben Sie sich das vorgestellt? Gut.

Und nun stellen sie sich vor, der Barde erzählt von dieser Schlacht und nichts passiert. Nur die Erzählung des Barden ist zu vernehmen, und Sie sehen nur wie er am Feuer sitzt. Welche der beiden Varianten würden Sie in einem Film bevorzugen?

Dachte ich mir. Nun, bei einem Spiel ist es ähnlich. Sie können die Story durch Zwischensequenzen und Dialoge erzählen. Allerdings sollte der Spieler nie das Gefühl haben, durch diese Elemente am eigentlichen Spiel gehindert zu werden.

Beeindruckende Zwischensequenzen haben ihren Platz in Spielen. Genauso ausgiebige Dialoge. Aber geben Sie Ihrem Spieler die Chance, die Dialoge zu überspringen, wenn er das möchte. Gibt es in den Dialogen wichtige Hinweise, dann sollten Sie alle Dialoge in einer Art »Notizbuch« festhalten, die als Gedächtnis der Spielfigur dienen kann.

Zwischensequenzen könnten über einen speziellen Eintrag im Hauptmenü betrachtet werden, sobald sie einmal erspielt wurden.

Wenn Ihre Animationen für die Zaubersprüche kleine Kunstwerke sind, dann will der Spieler sie sicherlich sehen. Aber wohl nicht jedes Mal. Auch hier sollte der Spieler lange Angriffs- und Zauberanimationen abbrechen können. Und mit »lang« meine ich alles, was über eine halbe Sekunde hinausgeht.

Sind die Angriffe und Animationen wirklich gelungen, dann könnten auch diese als Extra im Hauptmenü abrufbar sein (natürlich erst, nachdem der Spieler diese Angriffe auch freigespielt hat).

Eine gute Story ist sehr wichtig für das Spiel. Aber zwingen Sie sie dem Spieler nicht auf.

4 Spielbarkeit und Balance

Die interessanteste Story, die aufregendste Grafik, der bombastischste Sound und die umwerfendsten Charaktere helfen einem nicht weiter, wenn das Spiel zu leicht ist – oder zu schwer.

Gleichgewicht der Kräfte

Das richtige Verhältnis der Kräfte zu finden ist entscheidend für jedes Spiel. Stellen wir uns kurz ein Fantasy-Strategiespiel vor, in dem die Mächte der Finsternis gegen die des Guten antreten. Damit wir den Überblick nicht verlieren, beschränken wir uns auf drei Einheiten pro Seite.

Die grundlegende Einheit sind die Kämpfer. Sie sind recht schnell, und in erster Linie Nahkampfeinheiten. Auf der Seite des Guten könnten die Ritter diesen Part übernehmen. Auf der Seite des Bösen übernehmen Gargoyles diese Aufgabe. Sie sind nicht ganz so stark wie die Ritter, können aber Flüsse überqueren (da sie fliegen können) und sind auch etwas schneller.

Die Fernkampfeinheit sind Magier. Zu Fuß sind sie nicht gerade schnell, aber sie können sich zu bestimmten Punkten auf der Karte teleportieren. Im Nahkampf sind sie nicht sehr stark, aber auf einige Entfernung können sie mit ihren Zaubersprüchen beträchtlichen Schaden anrichten. Auf der Seite des Guten stehen die Priester für diese Einheit. Sie können sich jederzeit zu einem Tempel teleportieren. An offensiven Fähigkeiten haben sie heilige Blitze (ich weiß, nicht sehr originell). Die dunklen Fernkämpfer sind die Hexenmeister. Sie unterscheiden sich nur optisch von den Magiern. Ihre Rückzugspunkte für die Teleportationssprüche sind Pentagramme.

Auf beiden Seiten gibt es auch eine ultimative Einheit. Diese Giganten machen den meisten Schaden, bewegen sich recht schnell und können auch in einem etwas größeren Radius angreifen. Die »gute Variante« ist der Phönix. Seine Attacke verwandelt ihn in einen Feuerball, der kurzzeitig alles in seiner Umgebung verbrennt. Die dunkle Variante ist der Drache. Sein Feueratem geht nur in eine Richtung, hat dafür jedoch eine größere Reichweite als der Feuerball des Phönix.

Die Einheiten untereinander sind also recht ausgeglichen. Aber wie sieht es aus, wenn verschiedene Einheiten gegeneinander antreten? Wie viele Kämpfer brauche ich, um einen Fernkämpfer zu erledigen? Und wie viele, um einen Phönix oder Drachen zu erledigen?

Um das übersichtlich darzustellen und um möglichst einfach festzustellen, ob sich ein Spiel im Gleichgewicht befindet, gibt es die so genannte Gewinn- und Verlustmatrix. Dabei handelt es sich eigentlich nur um eine Tabelle, in der ähnlich wie beim Fußball die Ergebnisse der einzelnen Begegnungen festgehalten werden.

	Gargoyle	Hexenmeister	Drache
Ritter	1:1	1:10	0:40
Priester	10:1	1:1	1:20
Phönix	40:1	20:1	1:1

Tabelle 4.1: Gewinn- und Verlustmatrix

Ein Ritter ist also genauso stark wie ein Gargoyle, aber es bräuchte 10 Ritter, um einen Hexenmeister zu besiegen. Gegen einen Drachen braucht man schon 40 Ritter. Und bei der dunklen Seite sieht es genauso aus.

Wenn es jetzt keine weiteren Einschränkungen gibt, dann könnte man mit Drachen oder Phönixen das Spiel alleine bestreiten. Dann wäre die Zeit, die in die Grafiken und in den Quellcode für die anderen Einheiten geflossen ist, ziemlich vergeudet. Und auch der Spielspaß würde sich deutlich in Grenzen halten, da es eigentlich nur darauf hinausläuft, mit dem Drachen gegen den Phönix zu kämpfen. Was nun?

Eine der häufig genutzten Methoden ist es, die Erschaffung der Einheiten teuer zu machen. So könnte es deutlich länger dauern, einen Drachen zu beschwören, als es zum Beispiel dauert, 100 Ritter in den Kampf zu schicken. Dann wäre allerdings der Nutzen der großen Einheiten wieder weg.

Eine andere Methode ist es, den Einheiten spezielle Zusatzfähigkeiten zu geben, die aber nicht offensiv sind. So könnten Priester in der Lage sein, befreundete Einheiten zu heilen und Hexenmeister könnten besiegte Einheiten wieder auferstehen lassen, damit sie als Skelette weiterkämpfen.

Ausgleichen von Disbalancen

Am besten ist es jedoch, wenn wir eine Methode schaffen, die es erlaubt die großen Einheiten zu besiegen. So könnten wir eine weitere Einheit, den Helden, schaffen, die es erlaubt gegen die großen Einheiten anzutreten.

Auf der Seite des Guten könnte dies der Paladin sein. Auf der Seite des Dunklen ein Schattenkrieger. Die Namen sind hier nicht so wichtig. Wichtig ist, dass diese Einheiten gegen die großen Einheiten vorgehen können, jedoch gegen die kleineren Einheiten Probleme haben.

	Gargoyle	Schattenkrieger	Hexenmeister	Drache
Ritter	1:1	1:2	1:10	0:40
Paladin	2:1	1:1	1:6	1:1
Priester	10:1	6:1	1:1	1:20
Phönix	40:1	1:1	20:1	1:1

Tabelle 4.2: Gewinn- und Verlustmatrix mit Paladin und Hexenmeister

Nun wird es wichtig, dass wir in dem Spiel alle Einheiten einsetzen. Verlassen wir uns nur auf den Phönix oder Drachen, dann kann der Gegner mittels seiner Heldeneinheit diesen Plan recht schnell vereiteln. Jedoch ist dieser Held auch nur ein Mensch: Ungeschützt hat er kaum eine Chance gegen die Fernkampfeinheiten Priester bzw. Hexenmeister.

Doch nun sind die Kämpfer eine Einheit, die es nicht wirklich lohnt zu benutzen. Wir haben jetzt zwar dafür gesorgt, dass die Giganten nicht mehr dominierend sind, jedoch sind die Kämpfer weiterhin die schwächste Einheit.

Dies ist auf Abbildung 4.1 sehr gut zu sehen:

Wir können sehen, dass in der zweiten Variante ein Kreislauf zwischen Held, Gigant und Fernkämpfer entstanden ist. Jedoch ist der Krieger noch immer die eindeutig schwächste Einheit. Wenn wir auch im oberen Bereich einen Kreislauf wollen, dann müssen wir dem Krieger eine bessere Chance als dem Fernkämpfer geben.

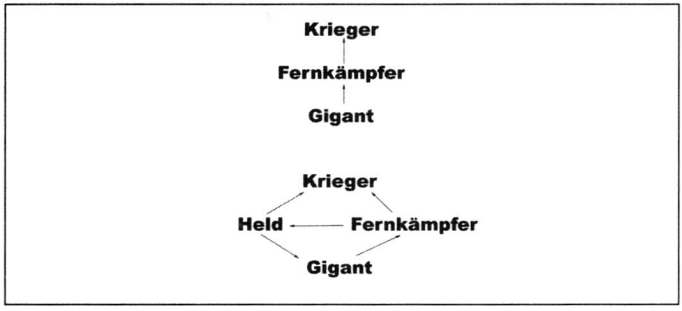


Abbildung 4.1: Visuelle Darstellung der Abhängigkeiten

Die Argumentation könnte sein, dass ein Ritter durch seine Rüstung deutlich langsamer ist, und aus diesem Grund ein leichteres Ziel für die Fernkämpfer ist. Der Krieger jedoch ist schnell genug, um an die Magiewirker heranzukommen bevor er ernstlichen Schaden erleidet.

Durch diese Änderungen haben wir nun ein ausgeglichenes System, in dem alle Einheiten gebraucht werden.

Es ist in der Regel besser, sich erst über diese visuelle Darstellung ein grobes Bild zu verschaffen, bevor man sich an die genaue Verteilung der Siegchancen macht.

In einem Rollenspiel könnte man auf diese Weise die Wirksamkeit von Magie oder von bestimmten Angriffen bestimmen. Nimmt man die klassische Einteilung der Magie in die Kategorien Feuer, Erde, Wasser und Luft vor, so kann man sich Feuer und Wasser als gleich stark ansehen. Ebenso sind Erde und Wind ebenbürtig. Also stellt man diese Elemente gegenüber, und erzeugt einen Kreislauf zwischen den jeweils anderen (siehe Abbildung 4.2).

Wenn man nun jeder Person und jedem Monster in dem Spiel ein Element zuweist, dann hat man auf eine leichte Weise etwas Strategie in die Kämpfe gebracht. Anstatt nun einfach den Gegner anzugreifen, macht es nun Sinn den richtigen Angriff zu finden, oder auf einen Zauberspruch auszuweichen.

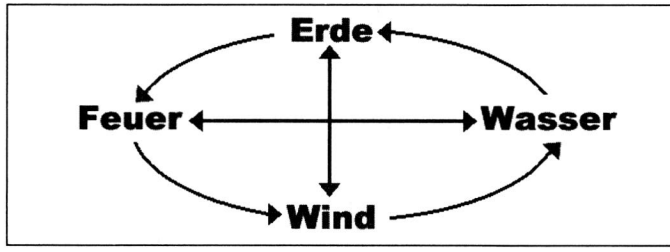


Abbildung 4.2: Die Abhängigkeiten zwischen den Elementen

Mit Hilfe der Abhängigkeitsgrafik und der Gewinn- und Verlusttabelle kann man dieses (oder auch jedes andere Konzept) beliebig ausbauen.

Haben wir bereits das oben genannte System mit den Elementen im Spiel, würden aber gerne noch etwas mehr Tiefe in die Kämpfe bringen, dann könnten wir zusätzlich zu den Elementen auch noch einen Typ festlegen. In einem Fantasy-Umfeld könnte man da zum Beispiel zwischen einer normalen Kreatur und Untoten unterscheiden. Eine untotes Monster könnte mit normalen Waffen nur sehr schwer zu besiegen sein, aber mit bestimmten Zaubersprüchen des Priesters oder heiligen Waffen sehr einfach zu besiegen sein.

Jedoch ist auch hier weniger oft mehr. Mit einem eingängigen, überschaubaren Konzept hat es der Spieler leichter sich im Spiel zurechtzufinden. Wenn er bei jedem Kampf in einer Tabelle nachschlagen muss, um einen passenden Angriff zu finden, dann sind die Chancen groß, dass er nicht lange bei diesem Spiel bleiben wird.

Kann er jedoch die Mechanik instinktiv verstehen, weil sie auf für ihn vertraute Konzepte zurückgreift, dann kann er spielen, anstatt sich zu überlegen wie man spielt.

Auch wird es einfacher für den Designer, das Spiel ins Gleichgewicht zu bringen. Eine zu mächtige Waffe am Anfang des Spieles vernichtet den Spielspass genauso wie zu starke Gegner. Die Stärke der einzelnen Kreaturen, Waffen und Zaubersprüche sollte in einem einfach zu lesenden (und ebenso einfach änderbaren) Format gehalten werden. Dies erlaubt es den Designern beim Testen des Spieles Änderungen am Gleichgewicht vorzunehmen, ohne den Programmcode ändern zu müssen.

Auch wenn Designer und Programmierer ein und dieselbe Person sind, ist es dennoch wichtig diese Änderungen schnell vornehmen zu können. In unserem Beispiel mit den Drachen und Paladinen haben wir gesagt, dass ein Priester es mit 10 Gargoyles aufnehmen kann. Dies klingt recht gut, aber wie stellen wir sicher, dass dies so ist? Ausgehend von der Beschreibung der Kreaturen haben wir zumindest Geschwindigkeit, Lebensenergie, Nahkampfstärke, Fernkampfstärke und Reichweite und Bereich des Angriffs. Und wenn man nun diese Werte so bestimmen möchte, dass das Ergebnis aus der Gewinn- und Verlustmatrix zustande kommt, dann ist dies keine leichte Aufgabe.

Wenn man nun für jeden Testlauf das Programm neu erstellen müsste, kann eine einfache Anpassung ein recht großer Aufwand werden. Auch wird es dann schwer die tatsächlichen Werte der einzelnen Einheiten im Überblick zu behalten. Gerade die Balance ist ein Punkt, der sich erst durch unzählige Testläufe einstellt. Das Testen sollte also ein einfacher Vorgang sein.

Level Design

Genauso wichtig wie ausgewogene Kämpfe ist ein ausgewogener Level. Stellen wir uns mal einen dunklen, unheimlichen Gang vor. Flackerndes Licht erhellt nur spärlich die karge Umgebung. Die Heldengruppe schleicht sich vorsichtig voran, die bitteren Erfahrungen mit den bisherigen Monstern hat sie jeden Übermut verlieren lassen. Endlich: Das scheinbar letzte Verlies, die Tür gesichert durch ein schwieriges Rätsel. Der erste Versuch die Tür zu öffnen resultiert in einer schweren Verletzung des Diebes. Ohne zu zögern benutzt der Priester den letzten Heiltrank, um den angeschlagenen Schlossknacker zu heilen. Endlich: Die Tür springt auf und die Helden betreten die Schatzkammer. Als sie die Truhe öffnen, finden sie... ein paar alte Socken.

So realistisch das auch sein mag (warum sollte diese Heldengruppe die erste sein, die sich auf die Suche nach diesem Schatz gemacht hat), genauso frustrierend ist es auch. Es ist einfach nicht fair, dass nach all dieser Anstrengung keine Belohnung auf die Helden wartet.

Nun, das Leben ist nicht immer fair. Aber ein Spiel sollte fair bleiben. Zumindest innerhalb seiner eigenen Regeln.

Nach einem anstrengenden Trip durch einen Dungeon sollte eine angemessene Belohnung auf die Helden warten. Dies könnte eine Rüstung sein, eine magische Waffe, eine Zwischensequenz oder einfach nur ein neuer Zauberspruch.

Auch sollte die Erscheinung einen Hinweis auf die Wichtigkeit geben. Wenn in einer einsamen Gegend ein bedrohlicher Turm steht, dann sollte darin besser etwas Interessantes sein (wenn nicht, hätte es auch eine Hütte getan). Der Versuch, die Umgebung interessanter zu machen, ist lobenswert, aber durch den Turm wird eine Erwartungshaltung geweckt. Wenn jetzt in diesem Turm nichts Interessantes ist, wird der Spieler enttäuscht sein. Das Spiel würde einen Teil seines Zaubers verlieren. Auch ist die Gefahr groß, dass ein anderes wichtiges Gebäude eventuell übersehen wird (»im letzten Turm war ja auch nichts«). Es muss ja nicht immer zu Kämpfen kommen. Wenn in diesem Turm ein alter, verschrobener Magier haust, könnte man in diesem Turm kleine Aufgaben erhalten, seine Tränke wieder auffrischen oder einfach nur ein Minispiel spielen.

Angemessene Level

Ich denke, wir sind uns einig, dass ein Spiel in erster Linie Spaß machen sollte. Erreicht ein Spiel dieses Ziel nicht, so ist es sehr unwahrscheinlich, dass es jemand für längere Zeit spielen wird. Nun ist es nicht immer leicht, den Finger auf die Aspekte zu legen, die den Spaß in einem Spiel ausmachen. Aber es ist normalerweise recht einfach, Fun-Killer zu finden.

Unübersichtliche Level und das »wohin muss ich nun gehen«-Syndrom sind für ein Spiel tödlich. Die einzige Entschuldigung für unübersichtliche Level ist die bewusste Entscheidung, ein Labyrinth in einem Spiel unterzubringen. Und auch dann sollte der Spieler nur einmal durch das Gewirr der Gänge müssen. Danach sollte der Held automatisch durch das Labyrinth auf die andere Seite (oder in das Zentrum des Labyrinthes) kommen. Hat der Irrgarten mehrere Ausgänge, dann sollte eine Übersichtskarte angezeigt werden, damit der Spieler den gewünschten Ausgang auswählen kann. Dies mag wieder einmal nicht realistisch sein, aber bei der Entscheidung zwischen Spielspaß und Realismus sollte immer der Spielspaß favorisiert werden.

Wenn die Bewohner eines von Monstern geplagten Dorfes dem Helden mitteilen, er solle zum »Glimmerpass« gehen, weil sich da der Eingang zur »Höhle des nicht ganz so großen Übels« befindet, dann sollte man

doch meinen, dass sich auch jemand finden sollte, der dem Helden den Weg dorthin erklärt. Es gibt wohl nichts Deprimierendes, als genau zu wissen wo man hin muss, aber den Weg dahin nicht zu finden. Dabei lässt sich dieses Problem recht einfach umgehen. Entweder einer der Anwohner erklärt dem Helden wirklich einfach nur den Weg (»Nimm den Pfad nach Osten bis zu den 3 Steinen. Dann geh nach rechts und über die Brücke. Du solltest den Eingang jetzt sehen können.«) oder man zeichnet den Eingang zur Höhle einfach in der Karte ein. Eine weitere Möglichkeit wäre ein vom Computer gesteuerter Charakter, der den Spieler zum Eingang der Höhle führt.

Ein weiteres typisches Übel sind zu weite Wege. Das typische Beispiel hierfür sind Hebel, die Türen in weit entfernten Räumen öffnen. Der Spieler verbringt die meiste Zeit damit von A nach B zu rennen. In der Regel befindet sich in den Räumen dazwischen auch nichts wirklich Spannendes- die Langeweile in diesem Teil des Spiels ist vorprogrammiert. Sobald sich eine solche Sequenz in einem Spiel befindet, kann ich förmlich hören wie sich der Level-Designer die Hände reibt: »Haha! Wieder eine Stunde mehr Spielzeit - und all das ohne große Anstrengung«. Allerdings wird auch der Spieler merken, dass hier nur Zeit geschunden werden soll.

Eine Variante dieses Themas ist der »Wettlauf gegen die Zeit«. Nicht weniger offensichtlich, kaum weniger nervig, aber etwas akzeptierter. Die Grundidee ist in etwa: Gehe von A nach B, lege da einen Schalter um und renn dann so schnell es geht wieder nach A. Auf dem Weg dahin sind ein paar Hindernisse, welche die Aufgabe etwas erschweren. Ist man zu langsam, dann heißt es die ganze Geschichte zu wiederholen. In manchen Varianten ist es nicht nur ein Hebel, sondern zwei, drei oder sogar noch mehr. Oder der Hebel ist eine Lampe, eine bewegliche Statue oder ein Schalter, den man über Umwege (Pfeilschuss, Magie oder eine mechanische Apparatur) betätigen muss.

Das Spiel im Fluss halten

Was ist falsch an diesem Bild: Tapferer Held dringt zum Drachenhort vor, besiegt den Drachen und verliert das Leben beim Rückweg aus der Höhle? Richtig, es ist einfach nicht fair. Und wie ich bereits weiter oben erwähnt habe: Unfaire Spiele sind bei Spielern nicht gerade beliebt.

Der Spieler sollte eine Möglichkeit haben sich schnell in die Stadt zu begeben und bei größeren Verliesen sollte es auch Abkürzungen hinunter

geben. Beliebte Möglichkeiten dies zu realisieren sind Zaubersprüche, die den Spieler in die Stadt zurückbringen; Teleporter, die, sobald sie einmal aktiviert wurden, direkten Zugang zu diesen Bereichen erlauben, Geheimgänge, die von der Höhle nach oben führen, und deren Aus- / Eingang von innen einmal geöffnet werden muss, dann aber stets offen bleiben.

Vermeiden Sie Sackgassen wann immer möglich. Insbesondere in Actionspielen sollte der Spieler durch den Level fegen können, ohne irgendwo anhalten zu müssen. In einem Puzzle- und Rätselspiel ist die Sachlage ähnlich. Zwar ist es in diesem Fall nicht so wichtig, die Spielgeschwindigkeit zu maximieren, aber eine Sackgasse veranlasst die meisten Leute in diesen Spielen dazu nach einem Ausgang zu suchen.

Umdrehen zu müssen hat immer auch einen kleinen Beigeschmack von Misserfolg. Nun kann man Sackgassen natürlich nicht immer vermeiden, aber wenn Sie es können, dann tun sie es.

Puzzles und Rätsel

Puzzles sind Spiele, bei denen der Schwerpunkt in erster Linie auf der geistigen Herausforderung liegt. Zumeist verlangen sie abstraktes Denken, Planungsvermögen und Kombinationsgabe.

Benutzt man Puzzles als Element in einem Spiel eines anderen Genres, dann besteht die Gefahr den Spielfluss zu unterbrechen. Das Puzzle ist »ein Spiel im Spiel«. Wenn ich mich durch eine Halle voller Monster gekämpft habe, das Klirren der Schwerter noch von den Wänden widerhallt und die Magie noch in der Luft knistert, ich mich dann um eine Ecke bewege, um dort von einem bärtigen, alten Mann ein Geometrisches Rätsel vorgesetzt zu bekommen, dann ist die Stimmung hinüber.

In den meisten Spielen ist es so, dass man den Mann erst passieren kann, wenn das Rätsel gelöst ist. Selbst wenn links und rechts von ihm noch genügend Platz wäre, um einfach vorbeizulaufen. Auch kann man den Alten meistens nicht angreifen. Das Rätsel wird dem Spieler vom Entwickler »aufs Auge gedrückt«.

Die bessere Lösung ist es, dem Spieler eine Wahl zu geben. Er kann versuchen den Rätselsteller anzugreifen. Und er sollte auch eine Chance haben den Kampf zu gewinnen. Zwar sollte der Schwierigkeitsgrad des Kampfes recht hoch sein – aber nicht unmöglich. Dies gibt dem Spieler

die Möglichkeit nach seinen Vorstellungen zu spielen. Warum sollte sich auch K'on-Ann, Kriegslord der unteren Ebenen, mit einem Puzzle abgeben? Insbesondere wenn die Axt in seiner Hand in etwa genauso groß ist wie die Person, welche die Frage stellt?

Eine weitere Möglichkeit ist es, die Puzzle und Rätsel in das Spiel zu integrieren. Rätsel, die auf Mechanik (Schalter, Pumpen, Hebel) oder auf Timing basieren, sind hierfür besonders geeignet.

Arten von Puzzles

Welche grundsätzlichen Arten von Puzzles kann man dem Spieler eigentlich stellen? Wie kann man die ganzen unterschiedlichen Puzzlearten kategorisieren? Welche Gemeinsamkeiten gibt es? Und welche Unterschiede? All diesen Fragen werden wir uns jetzt widmen.

Aus Sicht des Spieles kann man zwischen physikalischen/mechanischen und logischen/abstrakten Puzzles unterscheiden. Nun kann man zwar argumentieren, dass jedes mechanische Problem auch eine abstrakte Lösung hat, die dann nur noch umgesetzt werden muss. Dies ist natürlich richtig. Allerdings: Lässt man eine Kugel eine schiefe Ebene hinunterrollen, dann kann man dies zwar abstrakt durch mathematische Gleichungen beschreiben, es handelt sich aber trotzdem um einen physikalischen Vorgang.

Physikalische Puzzles

Die Grundlage eines physikalischen Puzzles ist im Endeffekt immer die Bewegung. Dinge müssen von einem Ort zum anderen bewegt werden, Schalter in die Richtige Position gebracht werden, Flüssigkeiten in ihrer Bahn geändert werden.

Kistenschieben

Wenn es ein klassisches Puzzle für Action Adventures und Rollenspiele gibt, dann ist es dieses. Die Grundidee ist recht simpel: Eine Reihe von Kisten (oder anderen Gegenständen) muss auf eine andere Position gebracht werden. Der Knackpunkt ist, dass die Kisten nur geschoben werden können. Kombiniert man dies mit einem engen Raum, in dem auch noch Hindernisse herumstehen, dann wird diese einfache Idee zu einer ziemlichen Herausforderung.

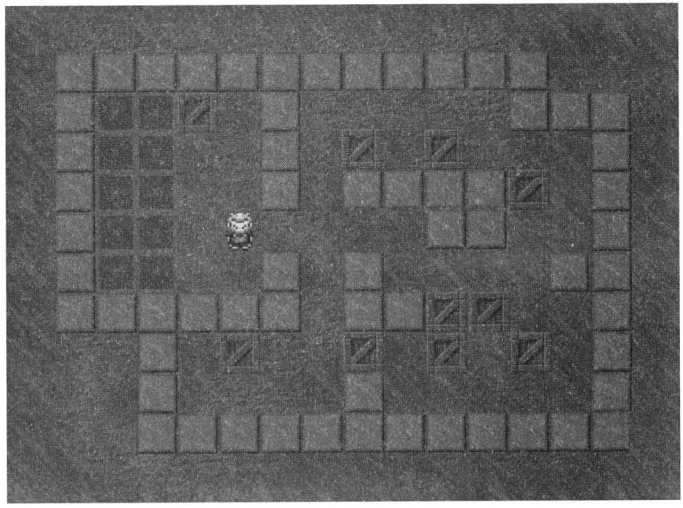


Abbildung 4.3: Ein Kisten-verschiebe Rätsel

Das eigentliche Aussehen dieses Rätsels kann beliebig variieren. In einem Fantasy-Spiel wird man wohl Steinblöcke oder Statuen auf in den Boden eingelassene Schalter schieben. In einem in der heutigen Zeit handelnden Spiel könnte man mit einen Räumfahrzeug schwere Gegenstände aufräumen. Ist es ein Superheldenspiel, so könnte der Held Tankschiffe aus dem Weg schieben. In einem Cyberspace Szenario müssen Datenpakete auf die Eingabe-/ Ausgabeports geschoben werden etc. Die Möglichkeiten der Umsetzung sind beinahe unbegrenzt.

Folgende Variationen bieten sich an:

- ✔ Vorgabe eines Zeitlimits
- ✔ Vorgabe eines Limits an Bewegungen
- ✔ Ziehen von Gegenständen
- ✔ Kombination von Ziehen und Schieben: Einige Gegenstände können nur geschoben, andere nur gezogen werden.

Natürlich lassen sich auch diese Variationen wieder kombinieren.

Schalterrätsel

Ein Labyrinth von Türen. Und jeder Menge Schalter. Betätigt man einen Schalter, dann öffnen / schließen sich mehrere Türen. Hinter den nun offenen Türen befinden sich weitere Schalter und (Überraschung!) Türen.

Diese Art von Puzzeln ist sehr eng verwandt mit Irrgärten. Und sie können genauso langatmig sein. In der Tat handelt es sich um eine Art Irrgarten mit beweglichen Türen. Und genauso wie ein Labyrinth sollten sie nicht an Orten eingesetzt werden, durch die der Spieler häufiger muss. Oder es sollte eine einfache Methode geben den Bereich schnell zu durchqueren, nachdem man es einmal auf die andere Seite geschafft hat.

Diese Art von Puzzle passt am besten in Schlossverliese oder in einen gut gesicherten Datenspeicher in einer Cyberspace-Welt.

Mögliche Varianten:

- ✔ In einigen Räumen sind Gegner, die beim Öffnen der Türen in den nächsten Raum gehen. Der Spieler kann nun entweder versuchen, diesen durch geschickte Schalterwahl aus dem Weg zu gehen, oder sich dem Kampf zu stellen.
- ✔ Kombination mit dem Kistenrätsel: Anstatt durch Hebel werden die Türen durch das Beschweren von Bodenplatten geöffnet. Je nach Positionierung der Bodenplatten erlaubt dies einige neue Varianten (ein Schalter in einer Ecke kann nur einmal betätigt werden. Ist die Kiste einmal darauf, kann sie nicht mehr von dort wegbewegt werden).
- ✔ Kombination mehrerer Schalter: Anstatt mit einem Schalter mehrere Türen zu betätigen, wird eine Kombination von Schaltern benutzt. Mit jedem Schalter, der neu dazu kommt, verdoppelt sich die Anzahl an Kombinationen. So sind sehr komplexe, aber auch sehr frustrierende Rätsel möglich. Mit zwei Schaltern lassen sich so zum Beispiel 4 verschiedene Türstellungen erreichen.
  - ✔ Schalter 1 oben, Schalter 2 oben
  - ✔ Schalter 1 oben, Schalter 2 unten
  - ✔ Schalter 1 unten, Schalter 2 oben
  - ✔ Schalter 1 unten, Schalter 2 unten

Teleporter-Puzzle

Auch hier handelt es sich um eine Variante des Labyrinthes. Aber anstatt durch Gänge und Räume zu irren, springt man hier mittels Teleporter von Raum zu Raum.

Ist in jedem Raum nur ein Teleporter, erhält man eine lineare Anordnung von Räumen. Bringt man nun in den Räumen mehrere Teleporter mit unterschiedlichen Zielen unter, dann steht der Spieler vor der Entscheidung, welchen er nun nutzen soll. Man kann ihm entweder keinerlei Hinweise geben (dann muss er einfach alle ausprobieren) oder die Portale durch Farben und / oder Symbole kennzeichnen.

Wichtig ist, dass es immer einen Weg zum Anfang zurück gibt. Der Spieler darf nie in einer Sackgasse landen oder den Eindruck haben, dass er aufgrund einer Entscheidung, die er willkürlich hat treffen müssen, etwas verpasst. Gibt es jedoch zu viele Wege zum Anfang zurück, dann dürfte ihn die Angelegenheit recht schnell frustrieren.

Ein beliebtes Konzept ist es, dem Spieler recht nah am Anfang eine Schatztruhe (oder etwas ähnliches) an einer Stelle zu zeigen, an die er so nicht herankommt. Er wird nun versuchen den Teleporter zu finden, der ihn in diesen Raum bringt (Das So-nah-und-doch-so-fern-Prinzip).

Teleporter lassen sich mit beinahe allen anderen Rätseln kombinieren. So kann ein Zugang zu ihnen durch Schalter freigegeben werden, oder sie werden erst aktiv, wenn die Kisten auf dem jeweiligen Platz stehen.

Türme von Hanoi

Im Großen Tempel von Benares, unter dem Dom, der die Mitte der Welt markiert, ruht eine Messingplatte, in der drei Diamantnadeln befestigt sind, jede eine Elle hoch und so stark wie der Körper einer Biene. Bei der Erschaffung der Welt hat Gott vierundsechzig Scheiben aus purem Gold auf eine der Nadeln gesteckt, wobei die größte Scheibe auf der Messingplatte ruht, und die übrigen, immer kleiner werdend, eine auf der anderen. Das ist der Turm von Brahma. Tag und Nacht sind die Priester unablässig damit beschäftigt, den festgeschriebenen und unveränderlichen Gesetzen von Brahma folgend, die Scheiben von einer Diamantnadel auf eine andere zu setzen, wobei der oberste Priester nur jeweils eine Scheibe auf einmal umsetzen darf, und zwar so, dass sich nie eine kleinere Scheibe unter einer größeren befindet. Sobald dereinst alle vierundsechzig Scheiben von der Nadel, auf die Gott sie bei der Erschaffung der Welt gesetzt hat, auf eine

der anderen Nadeln gebracht sein werden, werden der Turm samt dem Tempel und allen Brahmanen zu Staub zerfallen, und die Welt wird mit einem Donnerschlag untergehen.

Allerdings ist dies nicht wirklich eine indische Legende, sondern eine Geschichte, die sich der französische Mathematiker Edouard Lucas als Hintergrund für dieses Problem ausgedacht hat.

In der Tat ist es so, dass die Anzahl der zum Lösen des Problems nötigen Züge mit jeder neuen Zahl sprunghaft anwächst. Die korrekte Formel lautet

$$\boxed{AnzahlZüge = 2^{AnzahlderScheiben} - 1}$$

Für 3 Scheiben braucht man also 7 Züge. Für 4 sind es schon 15. Bei 7 Scheiben sind schon 127 Züge notwendig. Und dies, wenn der Spieler keinen Fehler macht.

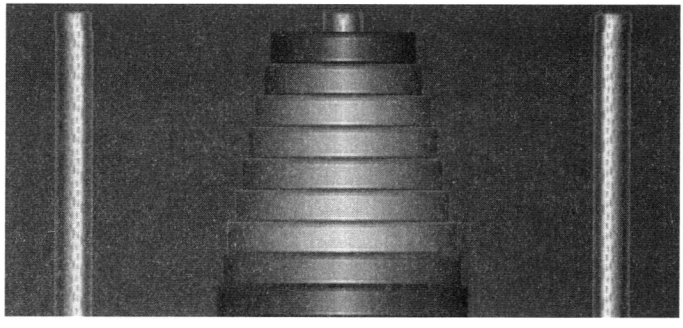


Abbildung 4.4: Die Türme von Hanoi

Wollen Sie die Türme von Hanoi in Ihrem Spiel einsetzen, so empfehle ich Ihnen, zwischen 3 (leicht) und 5 (schwer) Scheiben zu benutzen. Die 31 Züge, die dann auf der schwersten Stufe notwendig sind, um die Aufgabe zu lösen sollten genug sein.

In einem Spiel könnte der Held die Scheiben entweder direkt tragen, sie mit einem Gabelstapler (oder einem Arbeitsroboter) umschichten, oder die Scheiben mittels Magie von einer Säule zur anderen schweben lassen.

Objekt Puzzle

Diese Art von Puzzles basiert auf der Kombination oder dem Austausch von verschiedenen Gegenständen. In der einfachsten Version ist die Aufgabe, die Gegenstände in der richtigen Reihenfolge zu tauschen. Typische Sätze für diese Art von Puzzle sind »Natürlich bekommst du den Flux Kompensator – wenn Du mir dafür einen Plasmakonverter besorgst«. Und um den Plasmakonverter zu bekommen, braucht man einen Phaseninduktor. Diesen bekommt man nur im Austausch gegen... egal. Ich denke, das Prinzip sollte klar sein. Ein solches Puzzle zu planen ist recht einfach. Man braucht nur eine lange Liste mit Objekten. Diese verkettet man einfach, indem man die Reihenfolge festlegt, in der man die Gegenstände tauschen muss. Am Ende legt man fest, welcher Charakter welchen Gegenstand tauscht.

Variationen:

- ✔ Einer der Gegenstände in der Kette muss erst durch das Lösen einer Aufgabe gefunden werden.
- ✔ Man erstellt zwei oder mehr Ketten und überkreuzt diese dann.
- ✔ Man benötigt mehrere Objekte, um ein bestimmtes anderes zu bekommen (»Gib mir einen Stein der Macht und ein Silberamulett, und ich gebe Dir die Keule des Pythagoras«).

Eine Weiterführung dieser Idee ist, dass man bestimmte Objekte kombinieren kann. So könnte man eine Nadel, eine Leine und einen Stock zu einer Angel zusammensetzten.

Oder aus einem Seil und einem Haken einen Enterhaken basteln. Aus einem leeren Reifen wird dank einer Luftpumpe schnell ein aufgepumpter Reifen.

Natürlich funktioniert das auch in die andere Richtung: So könnte ein Funkgerät aus Sender, Empfänger, Antenne und Batterie bestehen. Man kann nun einzelne Teile dieses komplexen Gegenstands wieder mit anderen kombinieren. Es ist zwar einiges an Aufwand, eine Tabelle mit allen Gegenständen und ihren Kombinationsmöglichkeiten anzulegen, aber das Ergebnis lohnt sich. Neben der Verwendung für Puzzles kann der Spieler auch »einfach nur so« Gegenstände zusammenfügen, das Ergebnis dann wieder verkaufen, selbst benutzen oder einfach wieder neu zusammensetzen.

Die Welt des Spiels erhält mehr Tiefe, dem Spieler stehen mehr Möglichkeiten offen.

Abstrakte Rätsel und Puzzles

In einem abstrakten Rätsel wird dem Helden eine Aufgabe gestellt, die er nicht innerhalb des normalen Spielflusses (zum Beispiel durch das Umlegen eines Schalters) lösen kann. Ein typisches Beispiel für ein modernes Rollenspiel ist das »Tür-Code-Rätsel.« Das läuft normalerweise so ab: Spieler schleicht sich an eine Tür an. Bösewicht geht zur Tür und tippt einen geheimen Zahlencode ein. Der Held hört ein paar (unterschiedliche) Töne piepsen. Sobald die Luft rein ist, begibt sich der Held zur Tür, und wird da mit einer Grafik der Codetastatur konfrontiert. Durch Probieren erkennt er, dass jede Taste beim Betätigen einen anderen Ton erzeugt. Nun muss er »nur« noch auf der Tastatur die Melodie nachspielen, die er gehört hat, als der Bösewicht das Schloss bedient hat.

Doch nicht immer sind diese Rätsel so gut in das eigentliche Spiel integriert. Die einfachste Methode, ein beliebiges Rätsel in das Spiel zu integrieren ist einen Nichtspieler-Charakter darüber grübeln zu lassen: »Ich würde Ihnen ja gerne helfen, aber dieses Rätsel beansprucht meine ganze Zeit. Es muss doch eine Lösung geben...«.

Im Grunde lassen sich alle Rätsel und Knobeleien für diesen Zweck benutzen. Es gibt jedoch einige, die sich normalerweise besser in das Spielgeschehen einfügen lassen. Und auf diese wollen wir mal einen näheren Blick werfen.

Das klassische Rätsel

Der alte Mann, der die Brücke über den gähnenden Abgrund bewacht, sah die Gruppe streng an: »Wer über diese Brücke will gehen, muss gegen 3 Fragen bestehen!« Der Wächter, der der Gruppe ein Rätsel stellt, ist geradezu klassisch. Die Frage wird auf dem Schirm dargestellt, und der Spieler wählt eine der Antworten aus der Liste aus. Ist diese Antwort richtig, so darf er passieren. Antwortet er falsch, wird er normalerweise »hinausgeworfen«, d.h. er wird in einen anderen (aber nicht zu weit entfernten) Teil der Welt teleportiert.

Variationen:

- ✔ Zeitlimit
- ✔ Mehrere Fragen müssen beantwortet werden. Der Spieler bekommt keinen Hinweis, ob die einzelnen Fragen richtig oder falsch beantwortet wurden. Erst am Ende wird er fortgeschickt, wenn eine Ant-

wort falsch war. Dies macht es dem Spieler schwerer, alle Antworten zu testen.

Nur noch einmal als Erinnerung: Wenn ein NPC einen vor ein solches Rätsel stellt, dann ist es zu überlegen, ob man erlaubt ihn anzugreifen. Bei einem Schwierigkeitsgrad, der zwar heftig, aber nicht unmöglich ist, stellen wir den Spieler vor eine interessante Entscheidung: Entweder versucht er das Rätsel zu lösen, und riskiert damit sich ein weiteres Mal durch die Räume schlagen zu müssen, oder er riskiert den möglicherweise tödlichen Kampf mit dem Wächter.

Durch diese Wahl wird das Spiel echter, der Spieler kann seinen eigenen Weg gehen. Auch ist es eine großartige Sicherung: Falls das Rätsel zu schwer war, bleibt der Spieler trotzdem nicht stecken. Denn der Kampf steht im noch immer offen.

Musik- und Rhythmusrätsel

Das oben erwähnte Codeschloss-Rätsel zählt zur Gruppe der Musikrätsel. Diese lassen sich auf folgende einfache Formel bringen: »Auf die eine oder andere Weise muss eine Melodie nachgespielt werden«.

Ob man diese Melodie nun auf einer Codetastatur oder einer Flöte spielt, ist eigentlich nur Nebensache.

Ist es bei Musikrätseln meist nur nötig, die Musik irgendwie nachzuspielen, erfordern Rhythmusspiele Timing. Der Spieler könnte ein Nachricht mittels Morsecode verschicken müssen, einen bestimmten Code an eine Tür klopfen, oder gemeinsam mit den anderen einen Schlüssel drehen.

Logische Rätsel

Diese Rätsel verlangen in erster Linie logisches Denken. Streichholzrätsel sind ein sehr gutes Beispiel für diese Kategorie.

Legen Sie in Abbildung 4.5 vier der 36 Streichhölzer so um, dass drei unterschiedlich Grosse Quadrate entstehen.

Legen Sie in Abbildung 4.6 zwei Streichhölzer so um, dass 4 gleichschenkelige Quadrate entstehen.

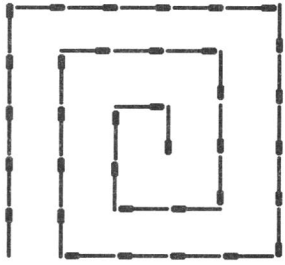


Abbildung 4.5: Streichholzrätsel

Abbildung 4.6: Noch ein Streichholzrätsel

Neben Streichholzrätseln gibt auch jede Menge anderer logischer Rätsel. Zum Beispiel dieses: Wie finden Sie aus 9 Kugeln die Kugel heraus, die etwas schwerer als die anderen 8 ist. Sie haben eine Waage, dürfen aber nur zwei Mal wiegen. Wichtig ist, dass Sie Rätsel wählen, die sich auch leicht am Rechner darstellen lassen. Es ist kein Problem, 9 Kugeln und eine Waage auf dem Schirm darzustellen, und dann dem Spieler zu erlauben, diese Kugeln auf die Waage zu legen.

Ein Rätsel, auf das man eine Antwort frei formulieren muss, oder bei der eine Multiple-Choice-Auswahl zu viel preisgibt, ist für ein Computerspiel eher nicht geeignet.

Es gibt von diesen Rätseln eine nahezu unerschöpfliche Anzahl, die Sie in Büchern, Zeitschriften und natürlich auch dem Internet finden können. Statten Sie Ihrer örtlichen Bücherei einen Besuch ab, und lassen Sie sich von den unzähligen Büchern zu diesem Thema inspirieren.

Minispiele

Minispiele sind kleine Spiele, die innerhalb des Hauptspiels für Abwechslung sorgen. Es handelt sich in der Regel um Spiele mit wenigen und einfach zu lernenden Regeln.

Spiele, die sich eignen, sind Kartenspiele und simple Arcade-Spiele. Sie lassen sich auch recht einfach in das Hauptspiel integrieren. Einfach eine Spielbank in einer Stadt positionieren, und schon kann man »einarmige Banditen«, Roulette und Kartenspiele problemlos in das Spiel integrieren. In einem modernen Spiel könnte eine Kart-Bahn das Umfeld für ein kleines Rennspiel liefern.

Und manchmal ist eine Erklärung auch nicht nötig. Spieler sind mit dem Minigame-Konzept vertraut und akzeptieren diese Spiele bereitwillig. So könnte in einem bestimmten Haus (oder Kerker) jede Tür zu einem neuen Minispiel führen. Allerdings ist es gut möglich, dass dies dann zuviel des Guten ist.

Minispiele sollten nur in Maßen eingesetzt werden, um den Spieler im Laufe des Abenteuers etwas Abwechslung zu gönnen, oder um eigentlich langweilige Aufgaben (wie zum Beispiel angeln oder Unkraut jäten) etwas spannender zu gestalten.

Casino- und Glücksspiele

Casinospiele lassen sich sehr schnell programmieren und lassen sich auch meist sehr gut in die Handlung einbauen. Auch bieten sie eine Möglichkeit, mehrere Spiele unter einem Dach unterzubringen. Genau aus diesem Grund sind sie in der einen oder anderen Form in vielen Spielen bereits vertreten.

Mögliche Spiele sind:

- ✔ Poker
- ✔ Baccarat
- ✔ Black Jack (17 und 4)
- ✔ Wettspiele
- ✔ Roulette
- ✔ Einarmige Banditen

Wenn man es dem Spieler erlaubt, innerhalb eines Casinos Geld zu gewinnen, so kann dies eine einfache Methode für den Spieler sein zu schummeln. Wenn er vor jedem Spiel speichert, und im Falle eines Verlustes einfach den alten Spielstand lädt, so kann man sich mit etwas Geduld ein kleines Vermögen zusammenmogeln.

Die einfache Lösung für dieses Problem ist es, den Spieler nicht innerhalb des Casinos speichern zu lassen. Muss er für jedes Mal neu laden einen langen Weg zurücklegen, dann kann man auf jeden Fall die Frequenz des Schummelns verringern.

Wettspiele sind einfach zu implementieren, bieten dem Spieler aber nicht viele Möglichkeiten für Interaktion. Meist wählt man zuerst seinen Favoriten aus einer Liste aus, dann läuft der eigentliche Wettkampf ohne Einflussmöglichkeiten für den Spieler ab. Am Ende wird dann dem Spieler ggf. sein Gewinn ausgezahlt.

Die Kartenspiele wiederum sind recht komplex, auch dauert eine einzelne Partie recht lange. Andererseits sorgen sie für ein authentisches Flair in der Spielbank. Die Automatenspiele und Roulette sind recht einfach von den Regeln. Der Spieler wird keine große Einführung brauchen, und eine einzelne Runde dauert auch nur ein paar Sekunden. Ich empfehle aus diesem Grund erst Roulette oder einen Einarmigen Banditen in Betracht zu ziehen.

Sportspiele

Die meisten Sportarten (sieht man mal vom Marathonlauf ab) eignen sich exzellent für Minispiele. Wettrennen, Ziel- und Kraftübungen lassen sich hervorragend konvertieren.

Bei der Steuerung gibt es zwei Varianten. Zum einen gibt es die »Rüttelspiele«. Bei diesen geht es darum, den Joystick möglichst schnell nach links und rechts zu rütteln. Spielt man mit der Tastatur, so muss man möglichst schnell abwechselnd die Pfeiltasten betätigen. Wird mittels Maus gesteuert, so muss diese meist in einem bestimmten Bereich schnell hin und her bewegt werden. Diese Art der Steuerung wird recht schnell auch für den virtuellen Sportler anstrengend. Aus diesem Grund bietet es sich auch an, sie bei Wettrennen (100 Meter Lauf) und Kraftvergleichen (Gewichtheben) zu benutzen.

Simuliert man Sportarten, bei denen die Technik im Vordergrund steht, so bietet sich eine auf Timing basierte Variante an. So könnte es beim

Hochsprung wichtig sein, bei jedem Schritt des Sportlers einen Knopf zu drücken. Bei richtigem Timing steigt die Anlaufgeschwindigkeit (und damit auch die Sprunghöhe). Nun nur noch im richtigen Moment zum Überqueren der Latte ansetzen (wieder mal durch Tastendruck) und schon ist der Sprung geschafft, oder die Hochsprunglatte gerissen.

Rüttelspiele:

- ✔ 100m Lauf
- ✔ Fahrradrennen
- ✔ Gewichtheben

Timingspiele:

- ✔ Hochsprung
- ✔ Weitsprung
- ✔ Rudern
- ✔ Hammerwerfen

Kombinationen:

- ✔ Kugelstoßen: Rütteln zum Kraftsammeln, dann Timing für den Abwurf
- ✔ Biathlon: Rütteln zum Rennen, Timing zum Schiessen

Sportspiele lassen sich zum Beispiel im Rahmen eines Wettkampfes oder eines Trainingswettbewerbes einsetzen. Die Disziplinen der jeweiligen Wettbewerbe sollten natürlich dem Zeitrahmen des eigentlichen Spieles entsprechen.

Die Belohnung könnten entweder Erfahrungspunkte sein (das Spiel wäre also eine Art Trainingsmodus), ein Geldpreis oder einfach nur der Ruhm und die Ehre. Welcher Spieler würde nicht gerne seinen Namen auf »Statue des Triumphes« eingraviert sehen?

Quizspiele

Ein Quiz-Minispiel hat einige Vorteile. Es ist leicht zu implementieren, kann sogar gegebenenfalls auf die normalen Dialog-Funktionen des Spieles zurückgreifen. Das Problem liegt allerdings in der Auswahl und Menge der Fragen. Geht man den einfachen Weg und stellt Fragen, wie sie

auch in einer normalen TV Quizshow auftreten können (»Wo wurden die Spaghetti erfunden?«), kann dies einen ziemlichen Bruch mit der Stimmung des Spiels bedeuten. Andererseits ist es recht schwer, genug Fragen zu finden, die zum Spiel passen. Damit Quizspiele auch nach mehreren Durchgängen noch Spaß machen, muss eine ausreichende Anzahl von Fragen vorhanden sein. Je mehr, umso besser. Mit weniger als 400 verschiedenen Fragen muss man gar nicht erst an den Start gehen.

5 Rollenspieldesign

In diesem Kapitel behandeln wir Rollenspiele. Was sie sind, was sie nicht sind. Was die Spieler an Rollenspielen interessiert und schließlich wie man ein eigenes System erstellt.

Rollenspieldesign bezieht sich sowohl auf den Entwurf von Abenteuern, als auch auf den Entwurf des Regelwerks. Obwohl der Schwerpunkt auf dem Rollenspielsystem liegt, werden wir uns auch dem Entwurf des eigentlichen Abenteuers widmen.

Was sind Rollenspiele?

Rollenspiele sind im Grunde ihres Herzens Simulationen. Mit einem mehr oder weniger komplexen Werk an Regeln, Tabellen und Ausnahmen versucht man, komplexe Abläufe wie Kämpfe nachzuahmen und die Eigenschaften und Fähigkeiten von Personen zu simulieren.

Der besondere Reiz liegt jedoch darin, dass man seinen Charakter selbst gestalten kann. Man hat Einfluss auf sein Alter Ego im Spiel. Dies schafft eine deutlich stärkere Bindung. Wenn man Stunden damit verbracht hat, seinen Charakter durch die Höhlen einer Fantasy-Welt zu steuern, viel Zeit damit verbracht hat, eine gute Ausrüstung zu finden, dann ist der Charakter auf einmal mehr als nur eine Ansammlung von Punkten auf dem Schirm.

Man fühlt sich für ihn verantwortlich, versucht ihn zu beschützen und möglichst gut auf die vor ihm liegenden Abenteuer vorzubereiten.

Rollenspiele sprechen den Jäger und Sammler in uns an. Wir jagen die Monster in einer unwirklichen Welt, und werden dadurch mit Erfahrungspunkten belohnt. Sobald wir genug Erfahrung gesammelt haben, steigt unser Charakter eine Stufe auf und kann neue Fertigkeiten erlernen und sich in vielerlei Hinsicht verbessern.

Doch auch die Ausrüstung, die es zu finden gilt, ist ein großer Anreiz. Rollenspiele strotzen gerade nur vor magischen, seltenen und wundersamen Waffen. Amulette der Stärke, magische Rüstungen und verzauberte Schwerter... es gibt nichts, was es nicht gibt. Und je seltener das Objekt, um so höher der Wunsch es zu besitzen.

Virtuelle Items wurden bei eBay schon zu enormen Preisen gehandelt. Dies zeigt ganz deutlich, welchen Wert die Spieler den seltenen Items beimessen.

Und es müssen nicht nur materielle Dinge sein, die man sammeln kann. Zauberer werden versuchen alle Zaubersprüche zu erlernen, Spieler von asiatischen Mönchen werden eventuell einen Ehrgeiz entwickeln alle Kampfstile zu meistern.

Das Rollenspiel-Genre

Die Frage, wann ein Spiel ein Rollenspiel ist, hat die Gemüter schon immer bewegt. Das Problem liegt auf der Hand: Es gibt kaum ein Spiel, in dem der Spieler nicht in die Rolle eines Helden schlüpfen muss. Das Einsammeln von Gegenständen und das Lösen von Rätseln sind inzwischen Elemente, die sich in beinahe jedem Spiel finden lassen.

Das typischste Element ist jedoch das Verbessern der Eigenschaften der Hauptfigur. Aber auch dies ist inzwischen keine Domäne der Rollenspiele mehr. Es gibt Ballerspiele, in denen man zwischen den Levels sein Raumschiff verbessern kann. Macht sie das zu Rollenspielen?

Strategiespiele haben inzwischen häufig »Helden«, die einen besonderen Stellenwert einnehmen. Meist sind sie stärker als andere Einheiten und direkt vom Spieler kontrollierbar. In einigen Spielen kann man auch die Fertigkeiten der Helden beeinflussen.

In Sportspielen gibt es häufig ein Trainingslager, in dem bestimmte Eigenschaften des Sportlers verbessert werden können. In Boxspielen können Stärke, Geschwindigkeit und Ausdauer trainiert werden. Tennisspiele erlauben es, Vor- und Rückhand sowie Aufschläge und Lobs zu trainieren.

Die Grenze zwischen Rollenspielen und »normalen« Spielen verwischt immer mehr. In den Online-Foren der Spielehersteller werden erbitterte Diskussionen darüber geführt, ob ein Spiel nun ein Rollenspiel ist oder nicht. Was für den einen ganz klar ein Rollenspiel ist, ist für den anderen ein Action Adventure. Für die einen ist es ein Weltraumballerspiel mit der Möglichkeit sein Schiff auszurüsten, für die anderen ein leicht zugängliches Rollenspiel.

Wo ist nun die Grenze? Gibt es überhaupt eine Grenze? Wenn ja, wer kann sie ziehen? Eine der einfachsten, aber auch treffendsten Definitio-

nen für ein Rollenspiel ist: »Ein Spiel ist dann ein Rollenspiel, wenn der Designer es als ein solches bezeichnet«.

Rollenspiel-System entwerfen

Wie geht man den Entwurf eines Rollenspiels an? Die Antwort lautet wie beinahe immer: Je nachdem. In einem actionorientierten System, in dem Klonk der Barbar durch die Gegend rennt, Türen eintritt und gegen Monster kämpft, reicht ein einfaches System. Je mehr Möglichkeiten wir Klonk bieten mit seiner Umwelt zu interagieren, um so komplexer wird unser System.

Für den Anfang beschränken wir uns auf ein einfaches, auf den Kampf fokussiertes System.

Fangen wir mit den grundlegenden Attributen an:

Attribut	Beschreibung
Stärke (ST)	Körperliche Kraft
Geschwindigkeit (GE)	Wie schnell bewegt sich der Charakter
Angriff (A)	Wie hoch ist die Chance zu treffen
Verteidigung (V)	Wie hoch ist die Chance nicht getroffen zu werden
Gesundheit (G)	Wie viel hält der Charakter aus

Tabelle 5.1: Attribute eines Charakters

Neben den rohen Werten spielt auch Glück und die »Tagesform« eine Rolle. Die beiden am häufigsten benutzen Varianten, um das Zufallselement zu berücksichtigen, sind Sammel- und Additionssysteme.

Das Additionssystem

Beim Additionssystem wird das Ergebnis eines Würfelwurfs zum Wert des Attributes gezählt. Für unser Testsystem gehen wir von den normalen 6-seitigen Würfeln aus.

Zuerst würfeln wir 5 mal, um die Werte von Klonk zu bestimmen. Mein Ergebnis dabei war 3, 1, 6, 4 und 2. Ein hoher Wert ist bei diesem System besser als ein niedriger Wert. Wir verteilen diese Werte auf die Attribute von Klonk, und legen dabei einen hohen Wert auf Stärke.

Damit Klonk auch einen Gegner hat, wiederholen wir das Prozedere, um Goridar den Furchtbaren zu erschaffen.

Attribut	Klonk	Goridar
Stärke	6	3
Geschwindigkeit	1	2
Angriff	4	3
Verteidigung	2	1
Gesundheit	3	4

Tabelle 5.2: Attribute von Klonk und Goridar

Bevor wir nun einen Kampf simulieren können, müssen wir noch festlegen, wie viele Trefferpunkte der Charakter hat (wie viel Schaden er aushält). Der Einfachheit halber sagen wir, dass die Anzahl der Punkte sich aus Stärke mal Gesundheit ergibt.

Klonk	6 * 3	18
Goridar	3 * 4	12

Tabelle 5.3: Trefferpunkte von Klonk und Goridar

Alle Werte im Additionssystem werden ermittelt, in dem man zum Grundwert das Ergebnis eines Würfelwurfs hinzuzählt. Um nun den Verlauf des Kampfes zu bestimmen, ermitteln wir zuerst, welcher von beiden anfangen darf. Klonk würfelt und hat eine 3. Goridar hat weniger Glück und würfelt eine 1. Damit nicht nur das Glück entscheidet addieren wir den jeweiligen Geschwindigkeitswert. So kommt unser Barbar auf eine 4 und Goridar auf eine 3.

Also greift Klonk zuerst an. Er würfelt eine 2, zählt seinen Angriffswert hinzu und hat ein Ergebnis von 5. Sein Gegner würfelt eine 3, addiert seinen Verteidigungswert (1) dazu und kommt auf 4. Der Angriff von Klonk trifft und macht 8 Punkte Schaden (2 gewürfelt plus 6 für die Stärke). Damit bleiben Goridar noch 4 Trefferpunkte.

Nun ist Goridar an der Reihe. Sein Angriffswert ist 4 (gewürfelt) + 3 = 7. Klonk verteidigt sich mit 3 + 2 = 5. Somit hat auch Goridar einen Treffer erzielt. Er würfelt eine 3 und macht somit 6 Punkte Schaden. Klonk ist damit auf 12 Punkte unten.

Nun ist Klonk wieder an der Reihe. Eine gewürfelte 6 macht auch diesen Angriff zu einem Treffer. Er würfelt noch mal und macht 5 + 6 = 11 Punkte Schaden. Damit verringert sich die Gesundheit von Goridar auf einen Wert von deutlich unter Null. Sieger: Klonk!

Das Sammelsystem

Während beim Additionssystem die Attribute zu dem Würfelwert hinzugezählt werden, werden beim Sammelsystem dem Attribut entsprechend viele Würfel geworfen.

Zum Angreifen würfelt Klonk 4 Würfel und erzielt 4, 2, 2 und 5. Damit ist jeder einzelne Wurf über dem Verteidigungswert von Goridar. Goridar würfelt mit einem Würfel und hat eine 3, liegt damit also knapp unter dem Angriffswert von Klonk. Somit ist die Anzahl der Erfolge für Klonk 4-0 = 4. Hätte er es geschafft den Angriffswert von Klonk zu übertreffen, hätte er die Anzahl von Klonks Erfolgen beim Angriff um eins reduzieren können.

Nun stellt sich die Frage der Schadensermittlung. Da der Barbar 4 Erfolge hat, kann man sagen, er hat es 4 mal geschafft seinen Gegner zu treffen. Er würfelt also 4 mal mit einem 6-seitigen Würfel. Und erzielt 14 Punkte. Damit hätte er Goridar mit seinem ersten Angriff besiegt. Offensichtlich brauchen wir also in diesem System deutlich mehr Lebenspunkte. Also ändern wir die Definition der Lebenspunkteberechnung. Nun gibt Stärke * Gesundheit nicht mehr die Anzahl der Lebenspunkte an, sondern die Anzahl der Würfel, die man zur Ermittlung der Lebenspunkte benutzt.

Klonk würfelt also 18 mal und kommt auf 62 Lebenspunkte. Goridar würfelt 12 mal und hat ein Ergebnis von 31 Punkten.

Ein weiterer Aspekt ist, dass in einem solchen System eine gewürfelte 1 immer einen Fehlschlag bedeutet. Also, wenn Klonk eine 1 als Angriffswert würfelt, wäre dies kein Erfolg, obwohl Goridar einen Verteidigungswert von 1 besitzt. Auch ist ein Wert von 6 immer ein Erfolg, unabhängig vom Wert des Attributs, gegen das gewürfelt wird.

Generell kann man sagen, dass in einem Sammelsystem die Anzahl der benötigten Würfelwürfe deutlich höher ist. Allerdings ist dies in einem Computerspiel nicht wirklich ein Problem, da auch die kompliziertesten Regeln in einem Sekundenbruchteil abgearbeitet werden können.

Welches System man nun benutzt, ist reine Geschmacksache. Ich werde im Rest des Buches ein einfaches Additionssystem benutzen, da dieses in der Regel deutlich einfacher nachvollziehbar ist.

Attribute und Fertigkeiten

Attribute sind die grundlegenden Eigenschaften eines Charakters. Stärke, Intelligenz, Ausstrahlung, Geschicklichkeit und magische Begabung sind die normalerweise genutzten Attribute.

Fertigkeiten beruhen auf der Anwendung von Attributen. So könnte ein Dieb die auf Geschicklichkeit beruhende Fertigkeit »Schlösser knacken« benutzen, um eine verschlossene Tür zu öffnen. Ein Barbar nutzt seine Fertigkeit im Kampf mit der Axt, um seine Gegner anzugreifen. Ein Kung-Fu-Kämpfer kann seine Fertigkeit im Adlerklauen-Stil benutzen, um den Gegner zu überwältigen. Andere Fertigkeiten sind Ringkampf, Entziffern von Schriftrollen, verschiedene Sprachen, Umgang mit den verschiedenen Waffen, Alchemie und Diplomatie.

Wenn es in einem System keine Fertigkeiten gibt, werden die für das Spiel grundlegenden Fertigkeiten, wie Angriff und Abwehr, auch als Attribute behandelt.

Ein auf Fertigkeiten basierendes System hat viele Vorteile. Wenn das Basissystem erst einmal steht, lässt sich beinahe alles als Fertigkeit behandeln. Ob nun Zaubersprüche, spezielle Angriffe oder das Mischen von Zaubertränken: All dies lässt sich als Fertigkeit abbilden.

Natürlich sind die Effekte beim Einsatz der einzelnen Fertigkeiten denkbar verschieden. Aber die Prinzipien beim Lernen, Verbessern und Anwenden der Fähigkeiten bleiben gleich.

Fähigkeitsbäume

Es gibt immer wieder Fähigkeiten, die auf anderen Fähigkeiten basieren. So könnte die Fähigkeit »Sprungtritt« nur dann erlernbar sein, wenn der Charakter bereits »Karate« auf einer Stufe von 3 oder höher hat. Andere Beispiele sind »Zaubertrank brauen«, welches Alchemie voraussetzt, und viele Zaubersprüche.

Ein Zauberspruch wie »Fliegen« beruht normalerweise auf Levitation (»schweben«) oder Telekinese. Manche Zauberfertigkeiten haben auch mehrere Voraussetzungen. Dies erlaubt es einem, das Erlernen mancher mächtiger Sprüche zu erschweren.

Wenn »Monster beschwören« direkt erlernbar ist, dann bedeutet dies, dass viele Spieler recht schnell in der Lage sein werden, sich im Kampf Unterstützung herbeizurufen. Dies könnte den Magiern einen sehr großen Vorteil gewähren. Also muss man eine glaubwürdige Methode finden, um diesen Spruch erst später zu gewähren. Nun ist es ja verständlich, dass ein Zauber, der Monster herbeiruft, die Fertigkeit »Monster kontrollieren« (der Spieler kann einem Monster befehlen jemanden anzugreifen) und »Beschwören« (ruft einen Geist herbei) voraussetzt.

Abbildung 5.1: Fertigkeitsbaum der Amazone in Diablo II

Manche Fertigkeiten können auch nur eine bestimmte Anzahl von Fertigkeiten aus einem oder mehreren Bereichen voraussetzen. So könnte die Kampffertigkeit »Eiserner Stand« drei andere Fertigkeiten der waffenlosen Kampfkunst voraussetzen.

Mit Hilfe solcher Bäume kann man eine nahezu beliebige Komplexität erzeugen. Wichtig ist nur, dass der Spieler immer die Übersicht bewahren kann.

Fertigkeitsnetze

Eventuell gibt es Fertigkeiten, die man auch benutzen können soll, ohne sie erlernt zu haben. Jeder kann versuchen einen Panzer zu fahren. Nur ist ohne jede Übung die Chance eines Fehlschlags recht hoch. Andererseits kann Erfahrung im Fahren anderer Fahrzeuge die Chancen wieder verbessern. (Es müssen keine Panzer sein. Wer auf einem Vogel Greif reiten will, sollte besser Erfahrung im Reiten anderer Tiere mitbringen. Gut, man kann es auch ohne jede Erfahrung versuchen – aber empfehlen würde ich das nicht).

Es gibt also Fertigkeiten, die anderen Fertigkeiten ähneln. Nun muss man noch eine Methode finden, diese Ähnlichkeit auch in Zahlen auszudrücken. Eine sehr übersichtliche Methode dafür sind Fertigkeitsnetze.

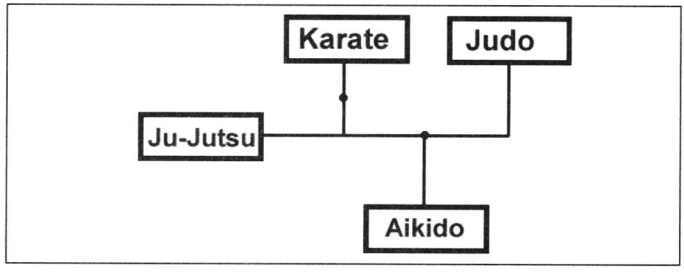


Abbildung 5.2: Fertigkeitsnetz Kampfsportarten

In dieser Abbildung sehen Sie den Zusammenhang zwischen den Kampfkünsten Karate, Judo, Ju-Jutsu und Aikido. Jeder Punkt in der Abbildung steht für einen Punkt Abzug auf dem Weg zu einer anderen Fertigkeit.

Ju-Jutsu vereint Elemente des Karate, Judo und Aikido. Aus diesem Grund kann man die Ju-Jutsu-Fertigkeiten mit einem geringen Abzug für Fertigkeitsproben in einer der anderen Sportarten nutzen. Karate und Judo unterscheiden sich jedoch stärker. Aus diesem Grund erhält man 2 Punkte Abzug, wenn man versucht, seine Karate Fertigkeit für eine Probe in Aikido zu nutzen.

Ein weiteres Bespiel für sich gegenseitig unterstützende Kenntnisse sind Programmiersprachen. Wer in C++ programmieren kann, wird keine großen Probleme haben Java Quellcode zu lesen. Perl Programmierer finden sich in PHP schnell zurecht, und selbst scheinbar weit auseinanderliegende Programmiersprachen wie Basic und Python haben noch genug gemeinsam, damit Wissen in der einen Sprache einen Vorteil beim Verstehen der anderen Sprache liefert.

Dieses Prinzip lässt sich natürlich beliebig anwenden. So könnte ein Zauberer mit sehr vielen Feuer-Zaubersprüchen eventuell Feuerzauber auch improvisieren können. Diese Entscheidung verändert jedoch das Gameplay sehr stark, da insbesondere das Erlernen und Testen der einzelnen Zaubersprüche einen Großteil des Reizes ausmacht.

Den Charakter verbessern – Level up

Eine der grundlegenden Ideen von Rollenspielen ist, dass der Charakter mit der Zeit besser wird. Dadurch, dass der Spieler seinen Charakter quasi aufzieht, entsteht eine starke Verbundenheit mit ihm. Viele Spieler tüfteln im Vorfeld lange herum, um eine für sie ideale Weise zu finden, den Charakter zu verbessern.

Wie nun dieses Lernen/Verbessern im Spiel präsentiert wird ist unterschiedlich. Man unterscheidet 3 große Bereiche, die sich wie immer beliebig kombinieren lassen. Im aufgabenbasierten Ansatz (zielbasiertes System) erhält der Charakter nach Beenden einer bestimmten Aufgabe (zum Beispiel die Befreiung der Prinzessin, das Finden des goldenen Amuletts etc) eine neue Fähigkeit hinzu, oder eine alte wird verbessert. Im Erfahrungspunkteansatz gewinnt der Charakter in jedem Kampf eine gewisse Menge an »Erfahrungspunkten«. Erreichen diese Punkte eine vorgegebene Schwelle, so erreicht der Charakter eine neue Stufe, und der Spieler kann die Fertigkeiten seines Charakters nach seinen Ideen verbessern. Im auf Fertigkeiten basierten System (Skill-System) verbessern sich die Fähigkeiten des Spielers durch die Anwendung derselben. Wer in erster Linie mit dem Schwert angreift, wird vielleicht stärker und er-

hält einen besseren Schwertangriff. Wer es liebt, durch die Macht der Magie mit Blitzen zu schleudern, wird besser in seinen häufig benutzten Zaubersprüchen.

All diese Systeme haben Vor- und Nachteile.

Das aufgabenorientierte System

Dieses System arbeitet nach dem Prinzip der Belohnung. Der Spieler löst eine Aufgabe und erhält dadurch eine Verbesserung seiner Fähigkeiten. Die Belohnungen sollten sich aus der Aufgabe ergeben.

Wenn der Spieler sich durch die Höhle der Eisriesen schlagen muss, um den gefangen gehaltenen Kriegshelden eines benachbarten Königreiches zu retten, dann könnte dies eine der folgenden Belohnungen nach sich ziehen:

- ✔ Der befreite Krieger bedankt sich, indem er dem Charakter des Spielers eine neue Angriffstechnik beibringt
- ✔ Der Endgegner in diesem Verlies lässt eine Schriftrolle fallen, durch die der Spieler einen Eiszauber lernt.
- ✔ Der Spieler findet einen Gegenstand, der eine seiner Attribute dauerhaft verbessert oder ihm neue Möglichkeiten eröffnet. Beispiele hierfür wären der »Gürtel der Stärke« oder ein Umhang der den Charakter kurzfristig unsichtbar macht.

Benutzt man magische Gegenstände, so müssen diese die Fähigkeiten dauerhaft erhöhen. Kann der Spieler die Gegenstände im nächsten Dorf wieder verkaufen, dann zählt dies nicht als eine den Charakter verbessernde Belohnung.

Ein gutes Beispiel für das Aufgabensystem ist »The Legend of Zelda: A Link to the Past«, welches ursprünglich auf dem Super Nintendo Entertainment System veröffentlicht wurde. Der Held des Abenteuers, Link, findet bei seinen Streifzügen durch die Verliese der Stadt Hyrule in Schatztruhen regelmäßig magische Gegenstände. Und zwar in der Regel immer einen Gegenstand pro Verlies. Und dieser ist dann auch der Schlüssel, um den Endgegner des jeweiligen Kerkers zu besiegen.

Diese Gegenstände, wie zum Beispiel einen Hammer, Handschuhe die seine Stärke erhöhen, Pfeil und Bogen und noch vieles mehr, werden zu einem integralen Bestandteil des Charakters selber. Er kann diese beson-

deren Gegenstände nicht verkaufen oder einfach nur liegen lassen. Sobald er sie in seinem Besitz hat, werden sie zu einem Teil von ihm. Jeder dieser Gegenstände markiert auch einen Teil der Entwicklung des Charakters, hat symbolische Bedeutung.

Wenn man in einem Abenteuer den Spieler kurz vor der finalen Schlacht ein sagenumwobenes Schwert aus einem Stein (oder Amboss) ziehen lässt, dann symbolisiert dies die Wandlung des Abenteurers zum wirklichen Helden.

Da man auf diese Weise als Entwickler des Abenteuers einen sehr großen Einfluss auf die Entwicklung des Charakters hat, ist Vorsicht angebracht. Einige Spieler werden wohl den Mangel an Entscheidungsfreiheit bemängeln oder eventuell sogar bezweifeln, dass es sich um ein »richtiges« Rollenspiel handelt.

Man kann dieses Problem mildern, indem man dem Spieler die Wahl zwischen verschiedenen Aufgaben lässt. Je nach Aufgabe erhält er eine andere Belohnung, und somit entwickelt sich der Charakter auch in eine andere Richtung.

Erfahrungspunkte-System

In diesem System wird davon ausgegangen, dass die Erfahrung, die ein Abenteurer auf seinen Reisen sammelt, messbar ist. Für jeden überstandenen Kampf erhält der Spieler einen gewissen Satz an Erfahrungspunkten (Englisch: Experience Points, XP). Erreicht der Spieler eine bestimmte Menge dieser Punkte, so erreicht er eine neue Stufe. Im Englischen wird die Stufe eines Charakters als »Level« bezeichnet, und das Erreichen einer neuen Stufe als »Level-up«.

Die Anzahl der Erfahrungspunkte, die ein Spieler erhält, sind zumeist abhängig von dem Schwierigkeitsgrad der Begegnung. So könnte ein Kämpfer der ersten Stufe 100 Punkte für einen Sieg über ein Skelett bekommen, aber ein Kämpfer der 5ten Stufe würde deutlich weniger Erfahrung erhalten. Die regeltechnische Erklärung ist meistens, dass ein hochstufiger Spieler in einem solchen Kampf einfach nicht mehr so viel lernen kann. Vom Standpunkt des Entwicklers aus ist es jedoch viel wichtiger zu verhindern, dass ein Spieler durch häufiges Kämpfen mit (zu) leichten Gegnern zu schnell »up-levelt«, da dies das Gleichgewicht des Spieles einfach zu stark verschieben würde.

Auch benötigt man in der Regel immer mehr Punkte, um auf die nächste Stufe zu gelangen. Eine der am häufigsten benutzten Verfahren, um die Menge an Erfahrungspunkten festzulegen, die für das Erreichen der nächsten Stufe notwendig ist, besteht darin, die Menge an zusätzlichen Punkten immer mit einem bestimmten Faktor zu multiplizieren.

Stufe	Benötigte Erfahrungspunkte	Unterschied zur vorigen Stufe
1	100	100
2	300	200
3	700	400
4	1500	800
5	3100	1600
6	6300	3200
7	12700	6400
8	25500	12800
9	51100	25600
10	102300	51200

Tabelle 5.4: Anstieg der benötigen Punkte bei einem Faktor von 2.0

Wem die bei ganzen Zahlen entstehenden Folgen zu offensichtlich sind, der sollte sein Glück mit einer krummen Zahl wie 2,5367 als Faktor versuchen.

Was nun genau bei dem Aufstieg auf eine neue Stufe passiert, liegt ganz beim Designer des Spieles. Die Attribute und Fertigkeiten können sich abhängig von der Charakterstufe verändern und dabei auf feste Formeln zurückgreifen (Stärke = 5 + Stufe * 1.5). Es ist aber auch denkbar, dem Spieler einfach nur eine gewisse Menge an Punkten zur Verfügung zu stellen und sie ihn frei verteilen zu lassen.

Mischformen erhöhen einige Attribute und Eigenschaften beim Level-Up, erlauben es dem Spieler aber, weitere Punkte nach Belieben zu vertei-

len. Oder Attribute »kosten« unterschiedlich viele Punkte. So könnte er zum Beispiel mit 2 Attributspunkten seine Stärke um nur eine Stufe erhöhen können (1 Punkt Stärke = 2 Attributspunkte) aber seine Geschwindigkeit um 2 Punkte (1 Punkt Geschwindigkeit = 1 Attributspunkt).

Diese Mischformen zwingen den Charakter zwar in eine gewisse Richtung, doch lässt sich dies durch die gewählte Charakterklasse begründen. Es macht durchaus Sinn, dass ein Kämpfer sehr schnell stärker wird, während der Zauberer deutlich mehr Attributspunkte investieren muss. Andererseits wird der Zauberer sehr viel einfacher Zauber lernen als ein Krieger.

Und das bringt uns zum nächsten Thema.

Charakter-Archetypen

Archetypen sind Urbilder, sie entsprechen allen Klischees, die wir von dem jeweiligen Charakter haben. Ein Ritter ist edel und stark, Magier sind alt und weise, Detektive schweigsam und mürrisch. Wenn Sie sich dieser Urbilder bedienen, dann fällt es dem Spieler leichter den Charakter einzuordnen.

Barbaren

Stark. Wirklich stark. Normalerweise benutzen Sie riesige Waffen und befinden sich am liebsten im Zentrum der Schlacht. Die Vorteile in der Muskelkraft gleichen Sie durch eine einfache Denkweise wieder auf. Barbaren treten Türen lieber ein, als das Schloss zu knacken. Probleme gehen sie meist sehr direkt (und schwer bewaffnet) an. Aufgrund der überlegenen Körperkraft können sie schwere Rüstungen tragen, halten eine Menge Treffer aus und verteilen jede Menge Schaden an die Gegner. Magie ist normalerweise nichts für den Barbaren. Typischer Satz »Klonk böse!«

Krieger

Krieger sind geübt im Umgang mit Waffen und Rüstungen. Zwar sind sie nicht so stark wie die Barbaren, aber Geschicklichkeit und viel Training machen Krieger zu einer erstklassigen Wahl, sollte es zu Kämpfen kommen. Zu den typischen Kriegern gehören Ritter, Samurai und Söldner.

Allen ist gemeinsam, dass sie sehr viel Wert darauf legen immer die besten Waffen und Rüstungen zu tragen. Typischer Satz: »Ich mag Schwerter!«

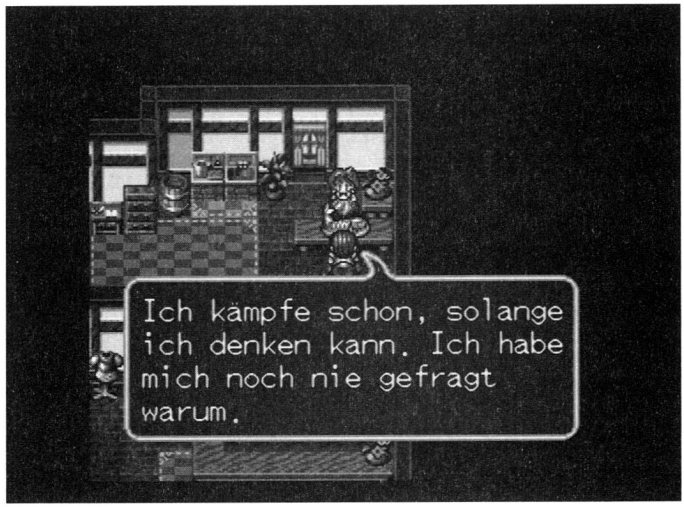


Abbildung 5.3: Der typische Kämpfer

Magier

Geheimnisvoll und weise. Meist etwas älter und mit rauschendem weißen Bart und einem langen Zauberstab ausgestattet. Der Magier befehligt die Mächte der Magie und wirft seinen Feinden Feuerbälle entgegen und wirkt Schutzzauber auf seine Freunde. Nahkämpfe sollte er tunlichst vermeiden, da die viele Zeit beim Studieren der magischen Bücher sich nicht gerade positiv auf seine körperliche Fitness ausgewirkt hat. Magier haben meist eine Abneigung gegen Rüstungen, da das Metall ihre Zauber beeinflusst, und das Gewicht ihnen zu schaffen macht.

Gauner

Sie nehmen es mit dem Gesetz nicht so genau und haben so einige Tricks auf Lager. Gauner sind darin geübt Fallen zu entschärfen und Schlösser

zu öffnen. Meist können sie entweder sehr gut mit dem Bogen umgehen und können auch kleinere Zauber wirken. Im Nahkampf können sie es zwar nicht mit Kriegern und Barbaren aufnehmen, aber hilflos sind sie auch im direkten Kampf Mann gegen Mann auf keinen Fall. Der Verlokkung einer Schatztruhe können sie sich meist ebenso wenig entziehen wie dem verlockenden Funkeln der Diamanten und Goldmünzen. Typischer Satz: »Oh, die Goldkette gehört euch?«.

Priester

Priester sind fromm und wirken Zauber, die sie von ihrem Gott gewährt bekommen. In einem Rollenspiel liegt ihre Hauptaufgabe meist darin, die Kämpfer zu heilen. Soll der Priester eine stärkere Rolle einnehmen, dann wird er meist zu einem Mittelding zwischen Krieger und Magier – mit Heilzaubern.

Paladin

Der Paladin ist ein »heiliger Ritter« – der Inbegriff alles Edlen und Guten. Er ist der Beschützter der Kinder und Waisen, rettet Katzen aus dem Baum und stürzt sich in jeden Kreuzzug, der griffbereit ist. Er hat leichte magische Fähigkeiten, die meist gegen Untote gerichtet sind, und kann entweder begrenzt heilen oder seine Kameraden mit Schutzmagie unterstützen. Aber seine eigentliche Stärke ist der Kampf.

Beschwörer

Der Beschwörer hat die Macht, Kreaturen zu seiner Unterstützung herbeizurufen. Er ist zwar körperlich nicht ganz so schwach wie der Magier, sollte aber trotzdem versuchen, an Kämpfen nicht direkt teilzunehmen, sondern lieber seine beschworenen Kreaturen vorschicken.

Charakter-Generierung

Es gibt prinzipiell 3 verschiedene Methoden, mit denen Spieler ihre Helden in einem Spiel erschaffen können:

- ✔ Vorgegeben – Es gibt nur einen, vorgegebenen Charakter, seine Laufbahn ist festgelegt.

- Klassensystem – Der Spieler kann bei Spielstart zwischen verschiedenen Charakter-Archetypen wählen.
- Freie Generierung – Der Spieler definiert seinen Charakter völlig über seine Werte, es gibt keine vorgegebenen Klassen.

Nun kann man diese Methoden auch mischen. So könnte man nur einen Helden haben, aber dieser kann seinen Beruf, seine »Bestimmung« nach kurzer Spielzeit wählen. Dies erlaubt es dem Spieler, die Klassen innerhalb des Spiels kennenzulernen. So könnten Repräsentanten der Klassen um den Spieler werben, versuchen ihn dazu zu bringen eine Laufbahn in ihrem Beruf einzuschlagen.

In Klassen- oder Freiensystemen könnte es vorgefertigte Charaktere geben, die der Spieler direkt auswählen kann. Dies erlaubt einen schnellen Einstieg in das Spiel, gibt aber den erfahrenen Spielern die Chance, sich einen Charakter nach Maß zusammenzubauen.

Soll man sich den Charakter von Grund auf erschaffen können, so bietet es sich an, ein Punktesystem zu verwenden. Zu Beginn hat der Charakter nur die Standardwerte in allen Kategorien. Wenn er sie verbessern will, dann muss er dafür mit »Erschaffungspunkten« bezahlen. Wichtigere Attribute sollten mehr Punkte kosten als Attribute, die im Spiel weniger häufig benutzt werden. Auf diese Weise kann der Spieler sich seinen Charakter so gestalten wie er es bevorzugt, es ist aber dennoch sichergestellt, dass der neue Charakter nicht übermächtig ist.

Bestehende Rollenspielsysteme

Wenn man sich nicht die Arbeit machen will, ein eigenes System zu entwerfen, gibt es eine Menge an freien Systemen, die man als Grundlage für ein Spiel benutzen kann.

Neben den mehr oder weniger frei verfügbaren Varianten der bekannten kommerziellen Rollenspielsysteme gibt es auch viele Regelwerke, die von Privatpersonen erstellt wurden. Es lohnt sich auf jeden Fall, einige Tage mit der Durchsicht dieser Systeme zu verbringen.

Auch wenn man sein eigenes System entwickeln möchte, so können einem die bestehenden Systeme sehr viele Anregungen bieten. Es gibt Hunderte von selbst erfundenen Zaubersprüchen, riesige Sammlungen von Fabelwesen und Monstern. Diese Sammlungen erlauben es einem, eine ungeheure Vielfalt in das Spiel einzubringen.

Ein eigenes System erschaffen

Wenn Sie ein eigenes System erstellen wollen, dann sollten Sie folgende Regel beherzigen: Machen Sie es nicht zu kompliziert. Komplexe System sind sehr schwer zu balancieren, und in den meisten Fällen wird der Spieler es nicht einmal bemerken, welche komplexen Berechnungen ablaufen, wenn er einen Gegner angreift. Was er wissen will ist, ob er getroffen hat, und wenn, wie gut der Treffer war.

Kampf und Magie

Kämpfe und der Einsatz von Magie gehören zu den Dingen in einem Computerrollenspiel, mit dem der Spieler die meiste Zeit verbringen wird. Aus diesem Grund sollte in die Planung des Magie- und Kampfsystems auch eine Menge Zeit fließen.

Kampf

Das einfachste Kampfsystem beruht nur auf zwei Werten: Schaden und Trefferpunkte (TP). Jeder Kämpfer macht Schaden, der von den Trefferpunkten des Gegners abgezogen wird. Der, der zuerst 0 TP erreicht, der verliert.

Doch wie definieren wir den Schaden den ein Charakter machen kann? Wenn es eine Zufallskomponente geben soll, dann ist es am besten, einen unteren und einen oberen Schadenswert anzugeben. Die bei Rollenspielern gebräuchliche Schreibweise dafür ist

$$Wert = nWm + q$$

n gibt die Anzahl der Würfel an, mit denen man würfeln muss.

m ist die Art des Würfels. Ein normaler, sechsseitiger Würfel wäre ein W6. Ein »Würfel« mit 100 Seiten ist ein W100 und so weiter.

q ist ein Wert, der nach dem Würfeln zum Ergebnis dazu gezählt wird.

Braucht man also einen Wert zwischen 1 und 6, dann würde man dies als 1W6 schreiben. Bei einem Wert zwischen 2 und 7 ist die Schreibweise 1W6+1.

Um nun die Unterschiede in der Kampfkraft der einzelnen Charakterklassen zu symbolisieren, könnte ein Magier 1W6 Schaden verursachen. Ein Beschwörer 1W10, der Priester 1W14, der Kämpfer 1W20 und der Barbar 1W25. Damit sollte der Barbar im Schnitt deutlich mehr Schaden anrichten als der Magier. Diesen Wert nennt man den *Schadenswürfel*.

Waffen können nun diesen Wert verändern. Ein Schwert könnte 6 Punkte Schaden zusätzlich machen, eine Axt 10 Punkte, während ein Dolch nur 2 Punkte Zusatzschaden verursacht.

Bis jetzt ist der Schaden also von der Charakterklasse und der Waffe abhängig. Rüstungen sollten nun den Schaden der von den Angriffen verursacht wird, verringern. Hierbei gibt es 3 Ansätze:

- ✔ Die Rüstung verhindert die Chance Schaden zu erleiden.
- ✔ Die Rüstung absorbiert eine feste Anzahl von Trefferpunkten.
- ✔ Die Rüstung absorbiert eine variable Anzahl von Trefferpunkten.

Im ersten Fall hat die Rüstung einen Wahrscheinlichkeitswert Schaden zu verhindern. Hat die Rüstung einen Wert von 33%, wird im Schnitt jeder dritte Schlag abgewehrt.

Im zweiten Fall hat die Rüstung einen festen Wert, den sie an Schaden abhalten kann. So würde eine Rüstung mit –5 Schaden von jedem Treffer 5 Punkte abwehren.

Im letzten Fall hat die Rüstung einen variablen Wert, ähnlich wie es bei den Angriffen auch der Fall ist. So könnte ein Kettenhemd 1W8+1 Trefferpunkte verhindern.

Wollen Sie es noch interessanter machen, dann geben sie der Rüstung selbst Trefferpunkte. Absorbiert sie Schaden, dann ziehen Sie diesen Wert von den Trefferpunkten der Rüstung ab. Geht die Anzahl der Rüstungstrefferpunkte auf 0 zurück, dann ist die Rüstung zerstört und kann keinen weiteren Schaden mehr aufhalten.

Sie können diese Werte auch kombinieren. Geben Sie doch den Rüstungen einen Chance den Schaden zu verhindern, und im Falle eines Treffers eine Anzahl von Punkten, die sie absorbiert. Leichte Rüstungen könnten eine bessere Chance zum Ausweichen haben, stärkere Rüstungen dafür mehr Schaden absorbieren und auch mehr TP haben.

Wir haben bisher einen Grundschaden, der von der Art des Charakters und der benutzten Waffe abhängt. Mit Hilfe einer Rüstung kann ein Teil des Schadens (oder auch alles) absorbiert werden.

Magie

Magie hat normalerweise eine dieser Ausprägungen:

- Angriffsmagie: Die TP des Verteidigers werden gesenkt. Entspricht in etwa einem normalen Angriff.
- Heilungsmagie: Die TP werden erhöht.
- Unterstützungsmagie: Erhöht den verursachten Schaden eines Kämpfers.
- Verteidigungsmagie: Verringert den erlittenen Schaden.

Jeder Zauberspruch hat normalerweise eine Erfolgschance, die abhängig vom Spruch und der Stufe des Zauberers ist. Hat der Spruch »Magische Rüstung« einen Schwierigkeitsgrad von 20%, so besteht eine 20%ige Chance, ihn zu verpatzen. Das Problem dabei ist nur, dass der Spieler es wohl nicht sehr gut finden dürfte, wenn ein Zauber mitten in der Schlacht gar keine Wirkung zeigt. Aus diesem Grund könnte man im Falle des Misserfolgs einfach die Wirkung des Zaubers abschwächen. Wird der Zauber nur knapp verpatzt, dann könnte er nur die Hälfte der normalen Wirkung zeigen. Wird er jedoch total verpatzt, dann hat er wirklich keine Wirkung. Um bei dem Beispiel der magischen Rüstung zu bleiben, könnte dieser Spruch einen Schwierigkeitsgrad von 20% haben, mit einer Patzerchance von 7%.

Um nun den Erfolg des Zauberspruchs zu ermitteln, wird mit einem W100 gewürfelt. Liegt der Wert über dem Schwierigkeitsgrad, ist der Zauber geschafft. Er zeigt dann seine volle Wirkung. Liegt er unter der Patzer-Chance, dann ist der Zauberspruch wirkungslos. Liegt er zwischen Patzer-Chance und Schwierigkeitsgrad, hat er nur die halbe Wirkung.

Beispiel: Dunkelzahn der Magier möchte den Spruch »Magische Rüstung« zaubern. Bei Erfolg hätte er eine Rüstung an, die 12W10+8 TP hat und pro Angriff 1W10+4 Punkte Schaden absorbiert. Der Schwierigkeitsgrad des Zaubers liegt bei 20%, Patzer-Chance 7%. Dunkelzahn würfelt eine 18. Damit ist der Spruch nicht perfekt gelungen, aber deutlich über der Patzer Chance von 7%. Die Wirkung des Zaubers wird halbiert, die Rüstung hat somit 6W10+4 TP und hält nur 0.5W10+2 Schaden auf. Hätte Dunkelzahn eine 6 oder weniger gewürfelt, wäre gar nichts passiert.

Ziele von Zaubern

Sie können die Zaubersprüche etwas interessanter gestalten, wenn Sie erlauben, dass ein Spruch entweder auf ein einzelnes Ziel oder auf alle Gegner oder Verbündete gesprochen wird.

Soll der Zauber »Heilung« auf alle Verbündete gesprochen werden, so erhält jeder einzelne einen Teil der Gesamt-TP gutgeschrieben. Dies erlaubt es entweder, einen einzelnen Charakter stark zu heilen, oder jedem der Mitglieder eine kleine Menge an TP zu geben. Wenn »Heilung« 1W20+10 TP heilt, und der Magier eine 8 würfelt, dann werden die 18 Gesamtpunkte gleichmäßig auf alle Verbündeten verteilt.

Nun ist es ja sicher nicht einfach, den Wirkungsbereich eines Zaubers auszudehnen. Aus diesem Grund könnte man den Schwierigkeitsgrad um 50% anheben. Hat der Heilzauber normalerweise einen Schwierigkeitsgrad von 10% bei einer Patzer-Chance von 2%, so würde sich der Schwierigkeitsgrad auf 15% erhöhen, wenn er auf alle Verbündeten gesprochen wird.

Kosten von Zaubern

Zauber kosten Energie, wenn man sie wirkt. Diese Energie wird normalerweise als Mana bezeichnet. Die Mana-Kosten eines Zaubers bleiben konstant: Auch wenn der Zauber verpatzt wird, wird die gleiche Anzahl an Mana-Punkten von den vorhandenen abgezogen. Ist das Mana bei 0, kann der Charakter nicht mehr zaubern.

Schwierigkeitsgrad von Zaubersprüchen

Ein Zauberspruch hat immer eine Stufe, die die Macht des Spruches angibt. Zauber der Stufe eins sind weniger mächtig als Zauber der Stufe 10. Normalerweise kann ein Magier einen Zauberspruch nur dann anwenden, wenn seine Stufe mindestens so hoch ist wie die des Zaubers. So könnte Grymlock, Magier der 4ten Stufe, alle Sprüche der Stufen 1, 2, 3 und 4 sprechen, aber keine Zauber der Stufe 5.

Einem Zauberer einer hohen Stufe sollte es leichter fallen, Zaubersprüche einer niedrigeren Stufe zu sprechen. Man könnte dies simulieren, indem man den Schwierigkeitsgrad senkt, wenn die Stufe des Magiers höher ist als die Stufe des Zaubers. Wenn man für jede Stufe, die der Zauberer höher ist, die Schwierigkeit um 20% senkt, dann hat sich der Schwierigkeitsgrad bei einem Unterschied von 3 Stufen halbiert.

	Stufe des Magiers									
Spruch-Stufe	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2	0,17	0,13
2		1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2	0,17
3			1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2
4				1,0	0,8	0,64	0,51	0,41	0,33	0,26
5					1,0	0,8	0,64	0,51	0,41	0,33
6						1,0	0,8	0,64	0,51	0,41
7							1,0	0,8	0,64	0,51
8								1,0	0,8	0,64
9									1,0	0,8
10										1,0

Tabelle 5.5: Änderung des Schwierigkeitsgrads eines Zaubers

Wenn Sie Ihren Spielern erlauben wollen, auch Zaubersprüche zu sprechen, die eigentlich zu schwierig für sie sind, dann sollten Sie dafür sorgen, dass Sie es auch wirklich schwer haben. So könnten sie den Schwierigkeitsgrad für jede Stufe Unterschied verdoppeln. Wenn Sie dem Spieler immer eine kleine Chance lassen wollen, dann können Sie für einen Schwierigkeitsgrad über 100% immer 99% angeben. Dann hat der Spieler immer eine kleine Chance, den Zauber doch noch zu schaffen.

	Stufe des Magiers									
Spruch-Stufe	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2	0,17	0,13
2	2,0	1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2	0,17
3	4,0	2,0	1,0	0,8	0,64	0,51	0,41	0,33	0,26	0,2

	Stufe des Magiers									
Spruch-Stufe	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
4	8,0	4,0	2,0	1,0	0,8	0,64	0,51	0,41	0,33	0,26
5	16,0	8,0	4,0	2,0	1,0	0,8	0,64	0,51	0,41	0,33
6	32,0	16,0	8,0	4,0	2,0	1,0	0,8	0,64	0,51	0,41
7	64,0	32,0	16,0	8,0	4,0	2,0	1,0	0,8	0,64	0,51
8	128,0	64,0	32,0	16,0	8,0	4,0	2,0	1,0	0,8	0,64
9	256,0	128,0	64,0	32,0	16,0	8,0	4,0	2,0	1,0	0,8
10	512,0	256,0	128,0	64,0	32,0	16,0	8,0	4,0	2,0	1,0

Tabelle 5.6: Schwierigkeitsgrades eines höheren Zaubers

Zusätzlich zur Anhebung des Schwierigkeitsgrads können Sie auch noch die Mana-Kosten erhöhen, oder einen Teil der Mana-Kosten von den Trefferpunkten und nicht von den Mana-Punkten abziehen. Es liegt bei Ihnen.

Das System erweitern

Unser System ist bisher sehr simpel gestrickt. Unser Charakter hat nur folgende Attribute:

- ✔ Stufe – Erfahrungslevel des Charakters
- ✔ Angriff – Wie hoch ist der verursachte Schaden?
- ✔ Trefferpunkte – Wie viele Treffer kann er aushalten?
- ✔ Magiepunkte – Wie viele Zauber kann er sprechen?

Die Waffe gibt einen Bonus auf den Angriff.

Die Rüstung hat 3 Werte:

- ✔ Rüstungspunkte – Die Trefferpunkte der Rüstung

- Chance auf Ausweichen / Blocken – Chance den kompletten Schaden zu vermeiden.
- Schadenverringerung – Wie viel Schaden geht auf die Rüstung?

Bis jetzt kann jeder Charakter jede Rüstung und Waffe benutzen. Wenn wir also einem Stufe-1-Magier das »Mächtige Schwert des Drachentöters« geben, das +50 Punkte Schaden macht, dann könnte er sich damit durch das ganze Abenteuer metzeln. Und das würde den Spielspass und die Balance deutlich trüben.

Wir müssen also einen Weg finden, die Benutzung der Waffen einzuschränken. Die einfachste Methode wäre es, jedem Gegenstand eine Stufenbeschränkung zu geben. Dann könnte das Schwert zum Beispiel erst ab Stufe 7 benutzt werden. Allerdings macht eine solche Beschränkung einen sehr künstlichen Eindruck. Besser ist es, die Beschränkung mit einem Attribut des Charakters zu verknüpfen. Das Schwert des Drachentöters ist sicherlich sehr groß, und verlangt deswegen nach einem starken Kämpfer, um es zu führen.

Führen wir also das Attribut Stärke in unser System ein. Um es uns einfach zu machen, können wir sagen, dass die Stärke zu Beginn die Hälfte des Schadenswürfels ist. Würfelt der Charakter also mit einem W20, dann hat er einen Stärkewert von 10.

Beim Stufenaufstieg könnte nun der Stärkewert automatisch erhöht werden, oder der Spieler verteilt Punkte auf alle Attribute.

Mit der Magie kann man es ähnlich machen. Gibt man dem Charakter ein Attribut »Magie«, dann kann man dieses als Ausgangspunkt für die Berechnung des Schwierigkeitsgrades nehmen. Somit steht es dann dem Spieler frei zu entscheiden, ob er lieber den Magiewert erhöhen möchte oder die Stärke des Charakters.

Selbst bei einem so einfachen System wie dem bisher entworfenen hat der Spieler jetzt schon einige Möglichkeiten. Er kann entweder die Trefferpunkte, die Stärke, den Magiewert oder den Mana-Wert seines Charakters erhöhen. Ein Zauberer mit viel Mana aber einer niedrigen Magie-Stufe kann viele schwache Zauber wirken, aber die Chance bei schweren Zaubern zu versagen ist recht hoch. Hat er einen hohen Magiewert aber nur wenig Mana, dann kann er zwar sehr souverän zaubern, aber die geringe Mana-Menge verhindert, dass er es zu häufig tut.

Bei den Trefferpunkten verhält es sich ähnlich. Je mehr TP ein Charakter hat, um so länger hält er im Kampf durch. Also wird jeder Spieler nach

Möglichkeit so viele TP wie möglich haben wollen. Was dann natürlich auf Kosten der Stärke und Magie geht.

Der Spieler muss also eine Entscheidung treffen. Und je nachdem wie er sich entscheidet, wird sich seine Art das Spiel zu spielen ändern.

Kampfablauf

Wir haben den Kampf bisher immer nur als eine Art Statistik betrachtet. Wie sieht es nun aber im Spiel aus? Werden die Aktionen Rundenweise gewählt, oder läuft der Kampf in Echtzeit ab? Wie sieht der Spieler das Kampfgeschehen?

Echtzeitkämpfe

Echtzeitkämpfe geben einem Rollenspiel ein Action Flair. Blizzards Diablo und Diablo II sowie die Zelda Reihe sind gute Beispiele für Echtzeitkämpfe.

Abbildung 5.4: Echtzeitkampf in Diablo II

Diablo ist mausgesteuert, die einzelnen Angriffe und Zaubersprüche werden erst ausgewählt und dann durch Rechts- oder Linksklicken ausgeführt. Bei Spielen, die über Tastatur oder Joystick gesteuert werden, kommt ein noch stärkeres Actiongefühl auf. Das Rollenspiel Terranigma für das Super Nintendo Entertainment System hat eines der ausgefeiltesten Echtzeitkampfsysteme. Der Charakter kann rennen, springen und angreifen. Diese Aktionen lassen sich kombinieren. So kann er im Rennen angreifen, im Sprung schlagen oder nach einem Sprung aus dem Anlauf angreifen.

Durch das Springen kommt noch eine Prise Hüpfspiel dazu. Das Ergebnis ist einzigartig. Wer noch ein SNES besitzt, sollte sich dieses Spiel unbedingt einmal ansehen. Es hat einige Elemente, die es wert sind, näher betrachtet zu werden.

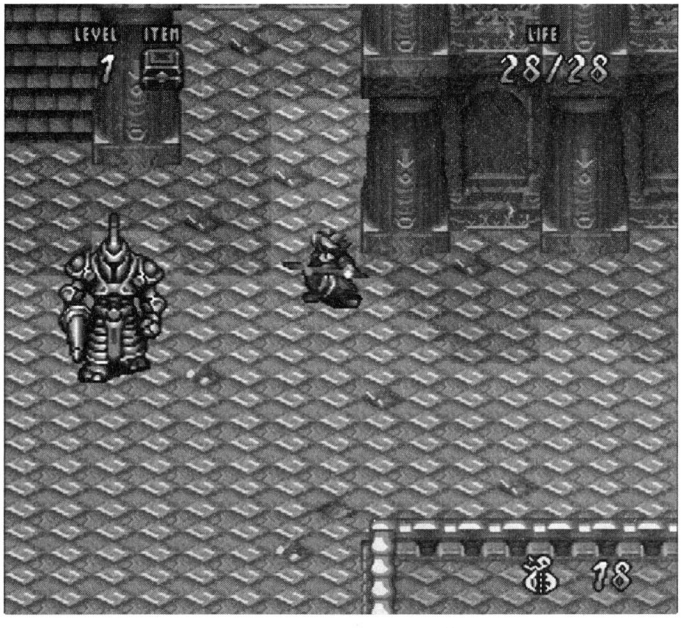


Abbildung 5.5: Terranigma – Echtzeitkämpfe mit Action Flair

Rundenbasierte Kämpfe

Bei einem rundenbasierten Kampfsystem kommt jeder der beteiligten Kämpfer nacheinander an die Reihe und wählt seine Aktion aus.

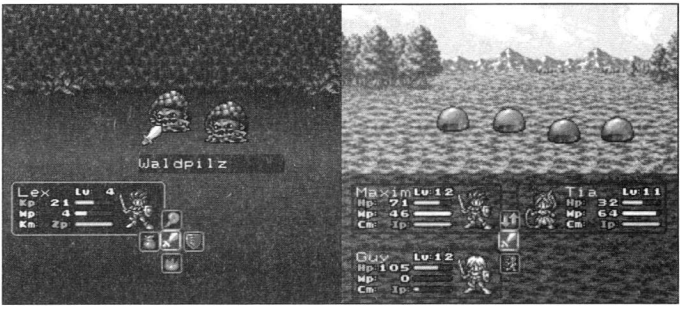


Abbildung 5.6: Rundenbasierte Kämpfe: Lufia II

Diese Entkopplung von Zeit und eigentlicher Aktion erlaubt es dem Spieler, eine ganze Gruppe von Abenteurern mit wenig Aufwand zu kommandieren. Es sind auch taktische Varianten möglich, wie das Ändern der Formation der Gruppe und das gezielte Auswählen einzelner Gegner.

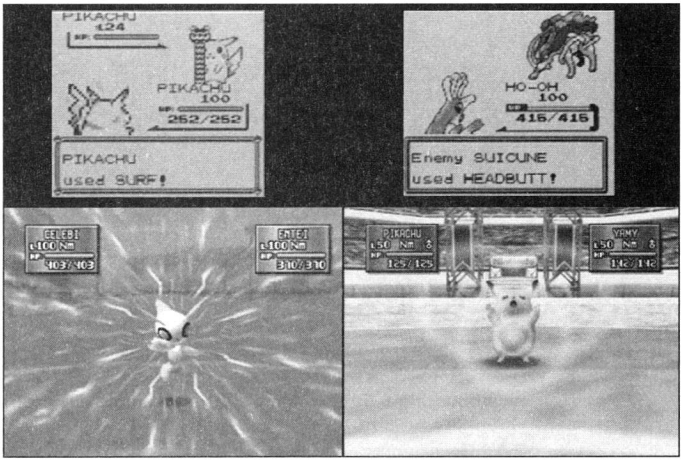


Abbildung 5.7: Pokemon auf dem Gameboy und dem Nintendo64

Ein weiterer Vorteil ist, dass sich das gleiche Spiel auf sehr unterschiedliche Weise darstellen lässt. So beruhen die rundenbasierten Kämpfe der Pokemon-Spiele auf den gleichen Regeln, nutzen aber die Möglichkeiten des jeweiligen Systems sehr gut aus. Sollten Sie zu den vielen Entwicklern gehören, die zwar gut programmieren können, aber Schwierigkeiten haben ein Grinsgesicht zu zeichnen, heißt das für Sie, das Sie das Spiel sehr leicht mir der berüchtigten Programmierer-Grafik entwickeln können und dann nach Fertigstellung des Spieles die Grafiken nach und nach verbessern können.

Active Time Battles (ATB)

Bei *Active Time Battles*, oder kurz ATB, handelt es sich um eine Variante der rundenbasierten Kämpfe. In diesem System hat jeder Spieler einen Geschwindigkeitswert. Dieser Wert bestimmt, wann er als Nächstes wieder an der Reihe ist.

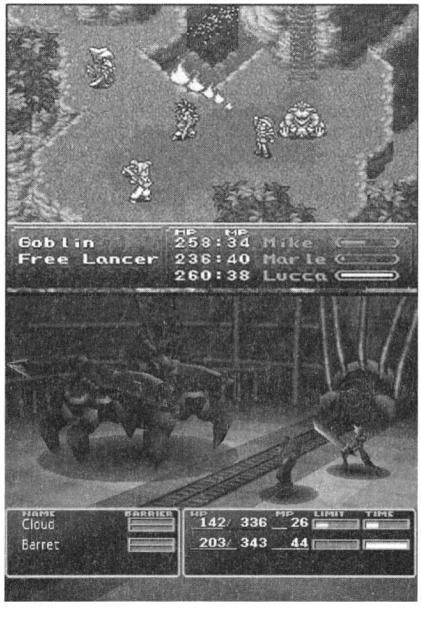


Abbildung 5.8: ATB-Kämpfe in Chrono Trigger und Final Fantasy VII

In jeder Runde wird eine Zeitleiste um einen Wert erhöht, der der Geschwindigkeit des Charakters entspricht. Sobald die Leiste voll ist, ist der Charakter am Zug. Auf diese Weise bleibt die Übersichtlichkeit des rundenbasierten Kampfes erhalten, es ist aber möglich, einzelnen Aktionen eine Dauer zuzuweisen. So könnte ein Charakter nach einem schweren Zauber erst 1.5 Runden später wieder an die Reihe kommen – wodurch sich natürlich die Abfolge der Aktionen ändert.

Zaubersprüche können Gegner verlangsamen und ihnen dadurch weniger häufig einen Angriff ermöglichen. Die Active Time Battles geben dem Spieler einige Möglichkeiten mehr, den Kampf zu planen, und der Entwickler kann sich voll und ganz mit der Spielmechanik austoben. Starke Gegner, die aber selten angreifen, und schwache, aber konstant attackierende Gegner sind zwei mögliche Extreme dieses Systems. Beim Entwurf der Gegner sind durch das Hinzufügen der Geschwindigkeit einige neue Optionen hinzugekommen.

Handel und Ökonomie

Neben den Kämpfen ist der Handel mit Gegenständen, Waffen und Rüstungen eine der wichtigsten Dinge in einem Rollenspiel. Der Kauf und Verkauf von Waren sollte auf möglichst einfache Weise möglich sein. Traditionell gibt es einen Zauberladen, einen Laden für Waffen und Rüstungen und einen »normalen« Laden in jeder größeren Stadt.

Die Geschäfte sind normalerweise so ausgelegt, dass der Spieler es möglichst leicht hat. Man kann meist in jedem Geschäft alles verkaufen, und die Dinge kosten überall gleich viel. Die Überlegung dahinter ist recht einfach: Sobald der Spieler einmal weiß, wo er Dinge billiger bekommt, wird er diese Sachen stets bei diesem Laden kaufen. Und es gibt eigentlich auch keinen Grund, warum man den Spieler zwingen sollte, an das andere Ende der Stadt zu laufen, nur um einen Gegenstand zu verkaufen.

Die Kehrseite der Medaille ist, dass sich einige Spieler über den mangelnden Realismus beklagen. Die Liste der Rollenspiel-Abgedroschenheiten führt diese Punkte zum Thema Geschäfte in Rollenspielen:

- ✔ Sobald man etwas in einem Laden verkauft, verschwindet es und taucht nicht im Angebot des Ladens auf.
- ✔ Jeder Laden hat immer geöffnet.
- ✔ Keine Stadt hat mehr als zwei Kaufleute, es sei denn, es ist wichtig, jeden einzelnen Kaufmann in der richtigen Reihenfolge aufzusu-

chen. Jeder dieser Läden führt jedoch die gleichen Waren zu genau dem gleichen Preis.

- ✔ Jeder Kaufmann kauft einem alles ab. Es ist kein Problem ein Schwert in einem Fischladen zu verkaufen.
- ✔ Alle Geschäfte, so exotisch sie auch sein mögen und wo immer sie sich auch befinden akzeptieren die gleiche Währung.
- ✔ Die Qualität der Waren ist nur abhängig von der Nähe des Endgegners. Selbst wenn das Spiel in einer riesigen Stadt startet, wird es in dem kleinen Dorf vor dem Eingang zur Höhle des Endgegners die besten Waffen geben.
- ✔ Kurz vor Ende des Spiels hat der Held einen enormen Ruf erlangt. Trotzdem wird keiner der Ladenbesitzer Waren kostenlos an den Helden abgeben, damit dieser die Welt retten kann.

Der erste Punkt erklärt sich durch den geringen Speicher der Konsolen, auf dem die meisten Rollenspiele laufen. Es ist meist zu aufwendig, den genauen Bestand der einzelnen Läden zu protokollieren und zu speichern. Auf einem PC ist dies kein Problem. Wenn Sie also mit diesem Punkt brechen wollen, dann tun Sie es. Es gibt dem Spieler vor allem auch die Möglichkeit, ein versehentlich verkauftes Stück wiederzuerlangen.

Warum haben die Geschäfte immer offen? Natürlich weil es dem Spieler normalerweise recht egal ist wie spät es in der Welt ist. Er will seine Gegenstände kaufen und verkaufen wann immer er will. Wenn es in Ihrem Spiel einen Tag/Nacht-Wechsel gibt, dann schließen Sie die Läden in der Nacht. Der Ladenbesitzer könnte dann noch einige Zeit in der örtlichen Kneipe zu finden sein und dann ins Bett gehen.

Für einen Entwickler mach es keinen Sinn, in einer Stadt zwei Läden zu haben, die genau das gleiche verkaufen. Auch ist die normale Stadt in einem Rollenspiel kaum groß genug, um zwei Geschäfte mit der gleichen Warenart zu rechtfertigen. Wenn Sie dennoch meinen, Sie müssten diese Regel widerlegen, platzieren Sie zwei Geschäfte in der Stadt, bei denen sich die Preise immer geringfügig unterscheiden. Ich bin allerdings der Meinung, dass 2 Geschäfte eher dafür sorgen, dass sich der Spieler über die extra Laufarbeit beklagt oder den zweiten Laden für überflüssig hält.

Die nächste Regel, »Jeder Laden kauft alles«, ist in der Tat nicht sehr realistisch. Zwar sehr bequem, aber nicht realistisch. Wenn jeder Laden nur noch die Gegenstände ankauft, die er auch wieder verkauft, dann muss sich der Spieler mehr von Händler zu Händler bewegen. Ein Kompro-

miss könnte sein, für ungewöhnliche Dinge den Preis anzupassen. Wenn Sie also einen Schwert im Zauberladen verkaufen wollen, dann werden Sie da halt etwas weniger Geld bekommen als im Waffenladen, da der Händler die Waren später an den eigentlichen Händler weiterverkauft.

Bei den Währungen sehe ich kein Problem. Aber da dieser Punkt in der Liste auftaucht, gehe ich hier darauf ein. Die Währung in einem Fantasy-Rollenspiel ist meist Gold oder ein anderes Edelmetall. Und da diese ihren Wert je nach Gewicht haben, ist die Art der Prägung doch eher unwichtig. Und in High-Tech-Umgebungen kann man davon ausgehen, dass Währungen nur noch eine untergeordnete Rolle spielen.

Dass es kurz vor dem Endgegner die beste Ausrüstung gibt, macht meiner Meinung nach durchaus Sinn. Allerdings könnte man ja beim Entwurf des Abenteuers dafür sorgen, dass die Stadt, die dem Endgegner am nächsten ist auch die größte/modernste Stadt im Spiel ist und aus diesem Grund auch die besten Gegenstände im Sortiment hat. Eine andere Erklärung wäre, dass der böse Herrscher die besten Waffenschmiede und Zauberkundige in seine Stadt befohlen hat, um seine Armeen auszurüsten.

Dass die Händler ihre Ware verschenken, wäre sicherlich revolutionär. Eventuell könnte man hier einen Zwischenweg wählen, und die Händler könnten Ihre Preise senken oder dem Spieler eine Waffe (leihweise) überlassen. Oder aber es bleibt alles beim Alten, und die Händler halten sich an die zweite Erwerbsregel: »Der beste Handel ist der, der den meisten Profit bringt«.

Ein weiterer Punkt ist der Warenbestand. Geschäfte haben meist einen unendlichen Vorrat an allen Gegenständen. Kauft man den Laden komplett leer, und geht kurz darauf wieder hin, ist der Bestand meist wieder gefüllt. Sie könnten den Laden stattdessen schließen, und einen Zettel an die Tür hängen, auf dem steht: »Geschlossen bis zum Eintreffen neuer Ware.« Wann diese neue Ware eintrifft, hängt ganz von Ihnen ab. Nach einer oder zwei Nachtruhen wäre eventuell ein guter Zeitpunkt.

Handeln mittels Drag'n'Drop

Das Shop System in Diablo II ist sehr einfach zu bedienen und gibt dem Spieler sehr viel Flexibilität. Da man in Diablo II auch sein Gepäck selbst organisieren muss, macht es durchaus Sinn, das Inventory des Ladens und das Inventory des Spielers nebeneinander zu positionieren.

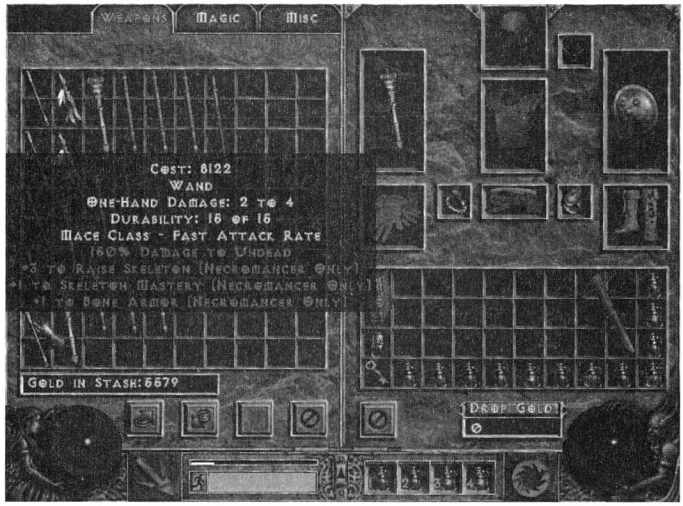


Abbildung 5.9: Diablo II Händler Interface

Der Spieler zieht die Gegenstände, die er kaufen möchte, einfach an die gewünschte Stelle im Inventar. Dadurch kann der Spieler direkt im Interface des Ladens seine Gegenstände ordnen.

Ein weiterer Vorteil ist, dass sich dieses System auch ideal dafür eignet, Gegenstände zwischen Spielern zu tauschen. Ein Aspekt, der bei einem Multiplayer Spiel wie Diablo II sehr wichtig ist.

Das herkömmliche Shop System

Bei einem herkömmlichen System wählt man erst aus einer Liste die Art des Gegenstands (Waffe, Rüstung etc.) und dann den Gegenstand und die Menge aus. Erst in einem zweiten Schritt werden die Gegenstände verteilt und die Waffen und Rüstungen angelegt.

Der Vorteil bei diesem System ist es, dass die Gegenstände sehr leicht von einem Charakter zum anderen übertragen werden können. Dieses System betrachtet die Gruppe als Einheit, jedes Mitglied der Gruppe kann stets auf alle Gegenstände und Zaubertränke zurückgreifen. Ein weiterer Vorteil ist, dass sich ein solches System nur mittels der Richtungstasten und zwei Aktionsknöpfen zum Bestätigen und Abbrechen bedienen lässt.

Auch benötigt es weit weniger Platz auf dem Bildschirm. Soll Ihr Spiel in einem Fenster laufen, dann ist geringer Platzaufwand ein wichtiger Punkt.

Beide Systeme haben Vor- und Nachteile. Welches sich besser für Ihr Spiel eignet, hängt von der Art der Steuerung (Maus oder Tastatur/Joypad) und vom Stil des Spieles selbst ab. Wie immer liegt die endgültige Entscheidung bei Ihnen.

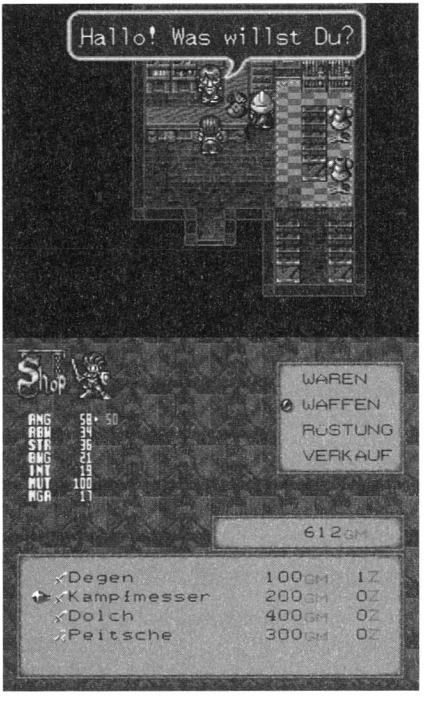


Abbildung 5.10: Das herkömmliche Shop System (Lufia II)

6 Das Design-Dokument

Das Design-Dokument beschreibt die Essenz des Spieles. Im Idealfall sollte man nach dem Lesen des Design-Dokuments in der Lage sein, die Ideen des Spieles zu begreifen. In ihm sind alle Aspekte des Spieles beschrieben, sei es nun der Ablauf, das Aussehen oder die Interaktion mit dem Spieler. Programmierer sollten aus dem Dokument entnehmen können, welche Grafik Engine sie brauchen und welche Datentypen sie anlegen müssen. Grafiker müssen nach der Lektüre des Dokuments in der Lage sein, sich die Welt, um die es geht, vorzustellen und Konzept-Grafiken anzufertigen (die dann wiederum ein Teil des Design-Dokuments werden). Musiker sollten beim Lesen des Dokuments die Stimmung des Spieles einfangen können, um passende Musik dafür zu komponieren.

Die Essenz des Spieles

In dem Design-Dokument sollte alles stehen, was ein Außenstehender braucht, um eine detaillierte Übersicht über das Projekt zu bekommen. Es entsteht meist aus einem Haufen von Skizzen und Notizen, wird dann kurzfristig zu einem ordentlichen Dokument, bevor es durch weitere Notizen und Skizzen ergänzt wird. Dieser Vorgang wiederholt sich, wobei ab einem gewissen Zeitpunkt keine neuen Ideen mehr hinzugefügt werden, sondern nur die bestehenden verfeinert und besser ausgearbeitet werden.

Grafiker haben neue Eingebungen und fügen Zeichnungen hinzu, die die Stimmung der Welt wiedergeben. Oder Beschreibungen der Klassen und Zaubersprüche werden konkretisiert und ausgebaut.

Wichtig ist, dass nicht nur das Ergebnis, sondern auch der Weg zu diesem Ergebnis dokumentiert wird. Werden Ideen nicht in diesem Spiel umgesetzt, so ist wichtig zu dokumentieren, warum dem so ist. Eventuell kann man sie ja im nächsten Spiel umsetzen?

Warum Sie ein Design-Dokument brauchen

In einem großen Projekt ist das Dokument eine Checkliste. Sie können überprüfen, ob alle geplanten Features auch wirklich im Spiel sind, und ob sie wie geplant funktionieren.

In einem kleineren Projekt hilft Ihnen das Design-Dokument sich auf die wesentlichen Dinge zu konzentrieren. Das Problem bei »kleinen Spielen« ist, dass sie dazu tendieren, nicht klein zu bleiben. Es gibt immer noch eine gute Idee, die es zu verwirklichen gilt, immer noch ein kleines Gimmick, das es zu implementieren gilt. Und so wird das Projekt mit kleinen Schritten immer größer und entfernt sich immer weiter vom geplanten Projekt. Ihr Freund, das Design-Dokument hilft Ihnen, sich auf das Wesentliche zu konzentrieren.

Früher oder später brauchen Sie vielleicht Hilfe bei der Entwicklung. Sei es beim Codieren oder beim Erstellen von Grafiken und Musik – irgendwo gibt es selbst bei den vielseitigsten Menschen eine Lücke, die es zu füllen gilt. Wenn Sie sich dann auf die Suche nach einem talentierten Menschen machen, der Ihnen helfen kann, ist das Design-Dokument Ihre beste Hilfe. Es zeigt dem potentiellen Helfer nicht nur, worum es eigentlich geht, sondern auch, dass Sie sich eingehend mit dem Projekt beschäftigt haben, dass es eben nicht nur eine Eintagsfliege oder eine schnelle Idee ist. Wenn Sie versuchen, in einem Internet-Forum für Pixel Grafiker nach Unterstützung zu fragen, ohne ein Design-Dokument vorweisen zu können, dann werden alle erfahrenen Grafiker entweder nicht reagieren oder nach einem Design-Dokument fragen.

Und in den Entwicklerforen wird es Ihnen auch nicht anders ergehen. Ein gutes Design-Dokument wird Ihnen in beiden Fällen einige Türen öffnen.

Und während diese Punkte sicherlich schon recht gut sind, ist die Tatsache, dass Ihnen ein Design-Dokument dabei hilft, Ihre Gedanken zu ordnen, und eine zentrale Sammelstelle für alle Ideen, Gedanken und Entscheidungen ist, sicherlich die wichtigste.

Wenn Sie nach 3 Monaten Arbeit fragen, warum sie damals festgelegt haben, dass Barbaren keine Motorsäge benutzen dürfen, sich aber nicht mehr daran erinnern können, dann werfen Sie einfach einen Blick in das Design-Dokument.

Elemente des Design-Dokuments

Ein Design-Dokument kann viele verschiedene Elemente umfassen. Nicht alle von Ihnen sind nötig, aber schaden kann keines von Ihnen. Ich würde Ihnen empfehlen, alle Punkte aufzulisten und während der Planung alle mit Inhalt zu füllen. Wenn bei dem einen, oder anderen Punkt

nicht ganz so viel zusammen kommt, ist das okay, aber nehmen Sie das nicht als Ausrede, ihr Design-Dokument zu vernachlässigen.

Die Übersicht

Das Design-Dokument beginnt mit einer Übersicht über das ganze Spiel in Kurzform. Stellen Sie sich vor, Sie müssten die gesamte Story und die wichtigsten technischen und spieltechnischen Features in einem Absatz zusammenfassen.

»Mjöllnir ist ein episches Rollenspiel, welches die nordischen Göttersagen mit High-Tech vermischt. Wir schreiben das Jahr 2030. Biotechnologie erlaubt es, den menschlichen Körper nahezu beliebig zu verändern und zu verstärken. Bei den Aushebungen zu einem neuen Gebäudekomplex im Norden Schwedens wird ein geheimnisvoller Hammer gefunden. Kurz darauf ereignen sich die seltsamsten Vorfälle: Zwerge und Riesen tauchen auf einmal auf, und Magie scheint wieder zu funktionieren. Der Spieler übernimmt die Rolle von Jord Godson, einem Interpol Ermittler. Im Laufe des Abenteuers muss er es mit Cyborgs, Göttern und wahnsinnigen Wissenschaftlern aufnehmen. Er findet die Liebe seines Lebens, verliert sie wieder und muss sich schließlich entscheiden, ob er auf der Seite der Menschen oder der der Götter steht.

Ein komplexes Kampfsystem, Dutzende von Zaubersprüchen und ein überwältigendes Arsenal an magischen und High-Tech-Waffen machen dieses Spiel zu einem Erlebnis«.

Scheuen Sie sich nicht, der Übersicht einen leicht reißerischen Unteron zu geben. Im Idealfall sollten Sie diesen Teil des Dokuments auch zu Werbezwecken benutzen können.

Die Benutzerschnittstelle

Alles, was in irgendeiner Form mit der Art und Weise zu tun hat, wie der Spieler mit dem Spiel interagiert, wird hier beschrieben. Vom Titelbild bis zum Abspann sollte hier alles aufgelistet sein. Auch die Übergänge zwischen den einzelnen Seiten müssen hier beschrieben werden.

»Das Titelbild bleibt für maximal fünf Sekunden sichtbar. Drückt der Spieler eine Taste, wird zum Menü Bildschirm umgeschaltet. Sind die 5 Sekunden um, und der Spieler hat keine Taste gedrückt, wird zufällig zum Tutorial oder Intro Animation umgeschaltet.«

Kann der Spieler auf einer Seite Einstellungen vornehmen, dann muss definiert sein, welche Auswirkungen das hat. Ein Beispiel: Der Schwierigkeitsgrad kann geändert werden, indem man einen Schieberegler von links (einfach) nach rechts (schwer) schiebt. Je nach Stellung des Reglers ändert sich auch das Gesicht des Charakters. Hat er bei »schwer« ein kantiges Heldengesicht, so sieht er auf leicht eher aus wie ein Zeichentrick-Baby. Die Übergänge sind fließend, obwohl es nur 4 verschiedene Bilder gibt, zwischen denen überblendet wird.

Die Personen

In diesem Teil sollten Sie alle wichtigen Personen des Spieles auflisten. Wenn Sie bereits Skizzen von den einzelnen Charakteren haben, dann gehören diese auch in dieses Kapitel. Der Leser dieses Abschnittes sollte in der Lage sein sich die einzelnen Charaktere auszumalen. Geben Sie eine kurze Übersicht über das bisherige Leben der einzelnen Leute und beschreiben Sie die Rolle, die dieser Charakter im Spiel übernimmt. Je wichtiger der Charakter, umso ausführlicher sollte die Beschreibung sein.

»Gustav Larsen ist der beste Cyberjockey, den Interpol je gesehen hat. Er ist leicht übergewichtig, versucht aber dem durch Gewichtheben entgegenzuwirken. Im Cyberspace benutzt er die Persona eines Shaolin Mönchs, und greift gegnerische Programme mit Kung-Fu-Attacken an.

Der Spieler übernimmt seine Rolle für eine kurze Zeit in Kapitel 2 des Spieles um die Daten aus dem Computer der Harakomi Gesellschaft zu beschaffen.«

»Neko ist ein Ninja und Chef der Konzernsicherheit von Harakomi. Er ist eine eher kleiner, drahtiger Mann. Er würde unscheinbar wirken, wären da nicht die toten Augen, die selbst den hartgesottensten Kämpfer noch einen Schauer über den Rücken laufen lassen würden. Neko ist nicht nur ein exzellenter Kämpfer mit dem Schwert, er kann auch mit allen modernen Waffen umgehen. Seit die Magie wieder erwacht ist, kann er durch Konzentration seine Abwehr stärken und sich für kurze Zeit unsichtbar machen. Neko ist der Endgegner von Kapitel 2. Sollte der Spieler es schaffen ihn zu besiegen, erhält er als neue Spezialwaffe einen Ninja Enterhaken.«

In diesem Stil sollten Sie alle Charaktere, die im Spiel vorkommen ausarbeiten.

Schnelldurchlauf

Stellen Sie sich eine Komplettlösung Ihres Spieles in einer Spielezeitschrift vor. Ungefähr auf diese Weise sollte der Schnelldurchlauf aussehen. Alle wichtigen Ereignisse, Gegenstände und Örtlichkeiten sollten beschrieben werden. Quests, die nicht zum Abschluss des Spieles notwendig sind, können und sollten kurz erwähnt werden. Allerdings ist es nicht notwendig sie in allen Details durchzugehen. Konzentrieren Sie sich in diesem Teil auf die elementaren Ereignisse.

Denn auch in diesem Fall wird dieser Teil einige Seiten in Anspruch nehmen. Anstatt alles auszuschreiben, können Sie auch Skizzen der Örtlichkeiten verwenden und in diesen die Puzzles und Gegner eintragen. In diesem Fall reichen ein paar kurze Hinweise über die Beschaffenheit der Umgebung und Stärke der Gegner. Im Falle von Rätseln und Puzzeln sollte die Art des Rätsels und die Lösung nicht fehlen.

Beschreibung der Klassen/Rassen und Berufe

Beschreiben Sie hier alle Charakterklassen, Rassen und Berufe, die der Spieler annehmen kann oder die in dem Spiel vorkommen. Erwähnen Sie neben den spieltechnischen Vor- und Nachteilen auch Hintergrundinformationen, die die Rolle der Klassen in der jeweiligen Welt beschreiben.

»Der Bogenschütze ist ein Bewohner des Waldes. Das Volk der Ellohoi lebt seit Jahrhunderten im Wald, da sie sich stark mit der Natur verbunden fühlen. Sie leben in kleinen Häusern, die sie in den Wipfeln der Bäume errichtet haben. In dieser Zeit haben sie sich ideal angepasst. Der typische Bogenschütze hat scharfe Augen, eine hohe Geschicklichkeit und eine leichte magische Begabung. Er fühlt sich in Metallrüstungen unbehaglich und trägt aus diesem Grund maximal eine Lederrüstung. Rastlose Ellohoi durchstreifen das Land auf der Suche nach Abenteuern.

Zu den Spezialattacken der Bogenschützen gehört der Mehrfachschuss, mit dem sie mehrere Gegner auf einmal angreifen können, das Analysieren des Gegners, welches die Anzahl der Hitpunkte und mögliche Schwachstellen offenbart und die Fähigkeit, Heiltränke aus Beeren und Kräutern herzustellen.«

So oder ähnlich könnte die Beschreibung der Bogeschützen-Klasse in einem Fantasy-Rollenspiel klingen. Stellen Sie sicher, dass die Unterschiede zwischen den einzelnen Klassen betont werden.

Erfahrungspunkte und Stufenanstieg

In diesem Teil beschreiben Sie das verwendete System, um die Erfahrungspunkte zu berechnen, die der Spieler nach einem gewonnen Kampf, dem Erfüllen eines Quests oder Lösen eines Rätsels bekommt.

Eventuell kann hier auch eine Tabelle der benötigten Erfahrungspunkte zum Erreichen der jeweiligen Stufe hilfreich sein.

Beantworten Sie in diesem Teil folgende Fragen:

- ✔ Wie viele Stufen gibt es im Spiel? Ist die Anzahl begrenzt oder nach oben offen?
- ✔ Wie viele Erfahrungspunkte bekommt der Spieler, wenn er einen Gegner besiegt, der weniger Erfahrungspunkte hat als er selbst? Die normale Anzahl an Punkten? Deutlich weniger? Gar keine? Abhängig vom Erfahrungspunkte-Unterschied?
- ✔ Wie viele Punkte bekommt der Spieler, wenn er einen sehr viel stärkeren Gegner besiegt? Die normale Anzahl an Punkten könnte dafür sorgen, dass der Spieler zu schnell aufsteigt.

Kampfsystem

Beschreiben Sie hier das Kampfsystem. Ist es rundenbasiert oder Echtzeit? Innerhalb der normalen Karte oder auf einem extra Bildschirm? Gibt es Zufallsbegegnungen oder sehe ich alle Gegner, bevor ich in sie hineinlaufe?

- ✔ Welche Waffen gibt es? Wie unterscheiden sich diese Waffen? Welchen Unterschied machen Nahkampf- und Fernkampfwaffen? Gibt es Voraussetzungen, die man erfüllen muss, um eine Waffe einsetzen zu können? Welche Wirkung haben magische Waffen?
- ✔ Welche Rüstungen gibt es? Welchen Unterschied gibt es zwischen den einzelnen Rüstungen? Voraussetzungen? Gibt es magische Rüstungen? Wenn ja, welchen Unterschied gibt es zu normalen Rüstungen?
- ✔ Kann sich die Gruppe formieren und die Formation auch beliebig ändern?
- ✔ Gibt es spezielle Angriffe? Wovon sind diese abhängig? Art der Waffe? Braucht man eine spezielle Fähigkeit, um die Spezialangriffe durchzuführen?

- ✔ Wie läuft der Kampf ab? Wie werden die einzelnen Angriffe ausgewählt?
- ✔ Sind manche Gegner immun gegen bestimmte Waffen?
- ✔ Richtet eine bestimmte Art von Waffen mehr Schaden bei bestimmten Gegnern an? (Das berühmte »Keulen gegen Skelette«-Beispiel)

Magiesystem

Gehen Sie kurz auf die Art der Magie in Ihrem Spiel ein. Woher bekommen die Magier Ihre Macht? Kanalisieren Sie Energie aus anderen Ebenen? Greifen sie auf von Göttern gewährte Mächte zurück? Ist die Magie immer gleich stark oder gibt es Gebiete, in denen ein Magier mehr Macht hat als in anderen? Gibt es Gebiete, in denen gar keine Zauberei möglich ist? Welche Arten von Magie gibt es? Sie können hier zwischen Gut und Böse, schwarzer und weißer Magie, Chaos und Ordnung unterscheiden. Oder sie weisen jedem Zauberspruch ein »Element« zu. Ist es möglich, dass in bestimmten Gebieten nur bestimmte Ausprägungen der Magie beeinflusst werden? So könnte ein Feuermagier auf einem Schiff mitten im Meer eventuell weniger gut zaubern können, da sein Element nicht in ausreichender Menge vorkommt, dafür aber das dem Feuer entgegengesetzte Element Wasser umso stärker ist?

Nachdem Sie diese Fragen geklärt und die Ergebnisse festgehalten haben, können Sie auf die technischen Details des Magiesystems eingehen. Welche Auswirkungen kann ein Zauberspruch haben? Wie wird er ausgewählt, wie wird er gewirkt, wie wird das Ziel bestimmt?

Und jetzt kommt der lustige Teil. Listen Sie alle Zaubersprüche auf. Je nach Art Ihres Spieles kann dieser Teil recht umfangreich werden. Angenommen, es gibt 10 Stufen an Zaubersprüchen, und jede Stufe hat 7 verschiedene Sprüche, dann sind das immerhin schon 70 Sprüche, die es zu benennen, beschreiben und auszuarbeiten gilt.

Gibt es verschiedene Arten von Magie oder Magiewirkern, dann vergrößert dies natürlich die Anzahl der Sprüche noch einmal.

Künstliche Intelligenz

In diesem Abschnitt erklären Sie, wie die einzelnen Gegner sich verhalten. Auf welche Weise sie patroullieren, worauf sie achten und wie sie kämpfen.

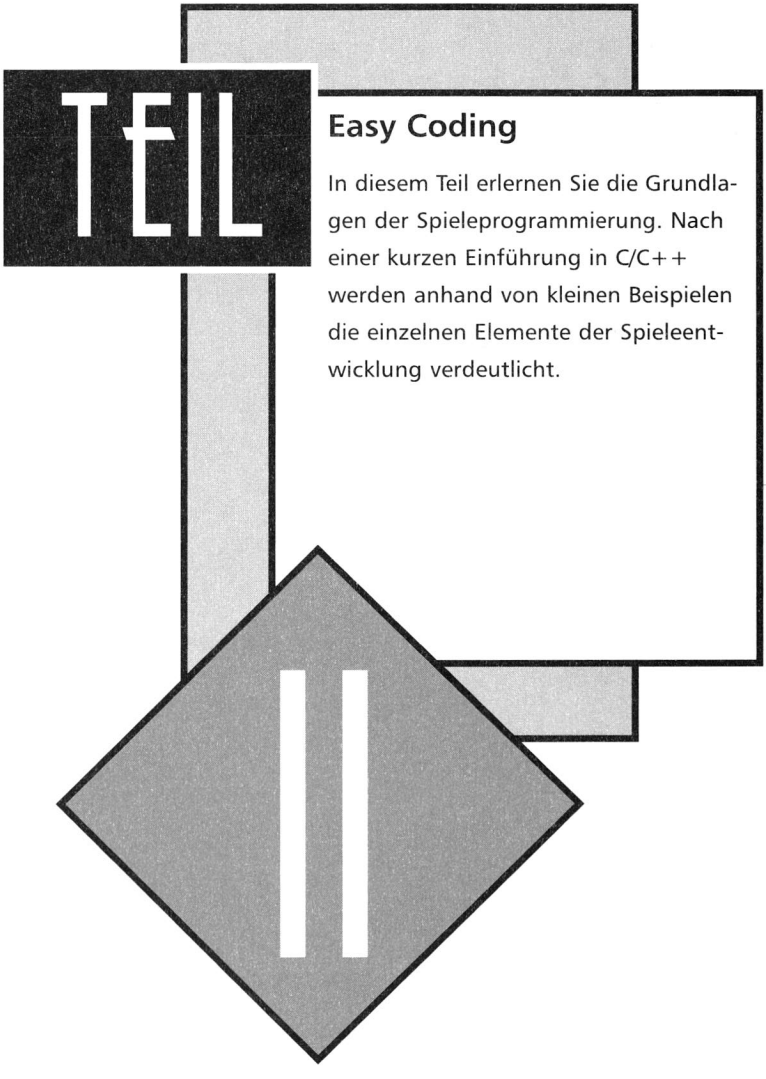
Erweiterungen des Design-Dokuments

Das Design-Dokument sollte über die komplette Entwicklungszeit erweitert und angepasst werden. Wenn es neue Grafiken gibt, die die Stimmung des Spieles wiedergeben, dann wären Kopien dieser Grafiken eine willkommene Bereicherung des Design-Dokuments.

Erweitern Sie die Beschreibung der Waffen und Zaubersprüche mit einer Liste der verwendeten Sound-Dateien und in welchen Situationen sie gebraucht werden.

Zumindest für Ihre ersten Projekte sollten Sie aber darauf achten, keine neuen Features hinzuzufügen. Konzentrieren Sie sich auf Ihre eigentliche Idee, und stellen Sie sicher, dass Sie das Spiel auch wirklich beenden. Zwar ist es verlockend, neue Features in das Spiel zu integrieren, aber Sie sollten widerstehen. Zu schnell wird man dazu verleitet, das Spiel mit immer weiteren Effekten und Elementen zu füllen.

Dies führt dazu, dass Ihr Spiel erst sehr viel später (wenn überhaupt) beendet wird. Sobald Sie sich sicherer fühlen und auch die Erfahrung haben, den Einfluss Ihrer Änderungen abzuschätzen, dann können Sie damit anfangen, auch die Feature Liste Ihres Spieles im Laufe der Entwicklung anzupassen.



Easy Coding

In diesem Teil erlernen Sie die Grundlagen der Spieleprogrammierung. Nach einer kurzen Einführung in C/C++ werden anhand von kleinen Beispielen die einzelnen Elemente der Spieleentwicklung verdeutlicht.

7 Grundlagen der Programmierung

Dieses Kapitel stellt eine Einführung bzw. Auffrischung der benötigten Kenntnisse der Programmiersprache C dar. Angefangen bei den einzelnen Kompilierungsschritten, über Makefiles, Zeiger und Zeichenketten wird ein Großteil der benötigten C-Grundlagen abgedeckt.

Die modernen IDEs (*Integrated Development Environment*) nehmen dem Programmierer eine Menge Arbeit ab. Leider verstecken sie auch vieles vor dem Programmierer. Macht dies bei einfachen Projekten noch keinen Unterschied, so kann es bei wachsendem Schwierigkeitsgrad (oder beim Wechsel der Umgebung) zu Problemen kommen.

Aus diesem Grund gehe ich hier noch mal auf die grundlegenden Konzepte wie Präprozessor, Compiler und Linker ein. Bevor die fortgeschritteneren nun die Augen verdrehen: Es wird nur ein kurzer Ausflug. Wer sich an seine kuschelige IDE gewöhnt hat, sollte eventuell trotzdem weiterlesen, da im Rahmen dieses Buches in erster Linie auf den GNU C/C++ Compiler eingegangen wird. Wer jedoch die (wenigen) Grundlagen verstanden hat, sollte jedoch keine Probleme haben, mit den gängigen IDEs und/oder Kommandozeilen-Compilern zurecht zu kommen.

Wem diese Materie jetzt zu trocken ist, kann auch einfach vorblättern und zu diesem Teil bei Bedarf zurückkehren.

Compiler, Linker und was man sonst noch alles braucht

Fangen wir mit dem klassischen Beispiel an:

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    printf("Hallo, ach du originelle Welt.\n");
}
```

Speichert man dieses Programm als »`hallo.c`" und kompiliert es dann mittels

```
gcc hallo.c -o hallo.exe
```

auf einem Rechner mit Windows Betriebssystem oder

```
gcc hallo.c -o hallo
```

auf einem Unix System, so laufen eigentlich 3 Schritte nacheinander ab:

- ✔ Der Preprozessor ersetzt die `#include` Anweisung durch den Inhalt der angegebenen Datei.
- ✔ Das Ergebnis wird kompiliert und dann
- ✔ mit den benötigten Standard-Bibliotheken gelinkt.

Präprozessor

Den Präprozessor (PP) stellt man sich am besten als so eine Art integriertes »Suchen und Ersetzen« vor, das noch ein paar extra Tricks kann, wie zum Beispiel Dateien einfügen, Bereiche in Abhängigkeiten von bestimmen Bedingungen ignorieren oder zulassen und komplexe Wortbausteine zu definieren.

Lassen Sie sich von dem Strich vor und nach FILE und LINE bitte nicht irritieren. Es handelt sich dabei um zwei Unterstriche, die programmbedingt zu einem langen Strich zusammengefügt werden.

```
#define DEBUG(f, text) fprintf(f, "%s, line %i: %s", __FILE__,
__LINE__, text)
```

Das Ergebnis von:

```
DEBUG(logFile, "Sprites erfolgreich geladen");
```

wäre dann zum Beispiel:

```
fprintf( logFile , "%s, line %i: %s", "test.c", 4,   "Hilfe" ) ;
```

wenn die `DEBUG` Anweisung in der Datei »test.c« in Zeile 4 stehen würde. `__FILE__` und `__LINE__` sind reservierte Bezeichner, die für die aktuelle Datei bzw. die aktuelle Zeile stehen. Sie werden meistens für Fehlerprotokolle oder ähnliche Aufgaben verwendet.

Weitere nützliche PP Bezeichner sind `__DATE__` und `__TIME__`, in denen als String das Datum und die Zeit der letzten Änderung verzeichnet sind. Hat man verschiedene Versionen seines Spieles verteilt, so können einem diese Informationen sehr häufig weiterhelfen:

```
#define DEBUG(f, text) fprintf(f, "%s, line %i: %s", __FILE__,
__LINE__, text)
#define VERSION  __DATE__ ", " __TIME__

DEBUG(stdout, "Version vom " VERSION);
```

Zwei aufeinander folgende Zeichenketten werden automatisch miteinander verkettet. Der Aufruf von DEBUG würde also zu folgendem Code-Fragment werden:

```
fprintf( stdout , "%s, line %i: %s", "test.cpp", 4,   "Version vom
Dec 16 2002, 21:48:26"  ) ;
```

Ein weiteres Einsatzgebiet sind die so genannten Include File Wächter. Ihre Aufgabe ist es, dafür zu sorgen, dass eine Header Datei nur einmal während des gesamten Präprozessordurchgangs beachtet wird. Ein Wächter sieht zum Beispiel so aus:

```
#ifndef __MY_GAME_HEADER__
#define __MY_GAME_HEADER__
/* Und hier kommen die gesammelten Header Infos */
#endif
```

Die `#ifndef` Direktive überprüft, ob der auf sie folgende Bezeichner noch nicht verwendet wird. Nur dann wird der Bereich zwischen `#ifndef` und `#endif` beachtet.

Also, wenn der Bezeichner `__MY_GAME_HEADER__` noch nicht verwendet wird, dann wird er über die `#define` Anweisung deklariert. Dies hat zur Folge, dass beim nächsten Bearbeiten des Headers der Bezeichner existiert. Wodurch der Bereich innerhalb des Wächters nicht noch mal eingefügt wird.

Natürlich gibt es zusätzlich zu `#ifndef` auch eine `#ifdef` Anweisung, die den Bereich nur beachtet, falls der angegebene Bezeichner existiert.

Sobald ein Projekt sich auf mehrere Dateien ausdehnt, sind diese Wächter absolute Pflicht. Mehr dazu in einem Augenblick.

Compiler

Der Compiler schnappt sich die Ausgabe vom Präprozessor und versucht das Ergebnis in Objektcode zu verwandeln. Objektcode ist vom Prinzip her ausführbarer Code, in dem jedoch noch einige Details fehlen. Im Beispiel oben ist zum Beispiel die Funktion `fprintf()` enthalten. Diese Funktion ist in der Header Datei `stdio.h` deklariert. Der eigentliche Code für diese Funktion findet sich jedoch in der Standard-C-Bibliothek, und wird nicht mitkompiliert. Die Funktionen aus der Standard-Bibliothek, welche vom Programm benutzt werden, werden erst beim Linken zum endgültigen Programm hinzugefügt.

```
gcc -c test.c -o test.o
```

Der -c Parameter veranlasst GCC dazu, das File test.c nur zu kompilieren (OK, eigentlich wird erst der Präprozessor aufgerufen und dann kompiliert). Der -o Parameter gibt den Namen für die Ausgabedatei an. Üblicherweise verwendet man .o als Erweiterungen für Objektdateien. Der GCC geht per Voreinstellung davon aus, dass diese Konvention eingehalten wird. Dies kann man beim Linken zum eigenen Vorteil nutzen.

Linker

Der Linker verbindet nun die Objekt-Dateien mit den Bibliotheken und erzeugt so ausführbare Programme. Allerdings werden nur die Funktionen, Strukturen und Variablen zum entgültigen Programm hinzugefügt, die auch tatsächlich gebraucht werden. Ängste, das Programm durch das Verwenden von Bibliotheken aufzublähen, sind in der Regel völlig unbegründet.

Wichtig ist, alle verwendeten Bibliotheken auch tatsächlich anzugeben. Die einzige Bibliothek, die man getrost vergessen darf, ist die Standard-C-Bibliothek. Diese wird vom Linker im Zweifelsfall ergänzt.

```
gcc -c test.c -o test.o
gcc test.o -o test.exe
```

Mit diesen Befehlen wird test.c erst kompiliert und dann zu der ausführbaren Datei test.exe gelinkt. Wir brauchen beim Linken keinen Parameter anzugeben, da das Erstellen eines lauffähigen Programms der Normalfall ist und deswegen keines besonderen Parameters bedarf. Da der Linker an der Endung .o erkennen kann, dass es sich bei der angegebenen Datei um ein Objekt File handelt, ist auch hier eine weitere Kennzeichnung überflüssig.

Nachdem wir nun Dateien durch den Präprozessor jagen, kompilieren und linken können, stellt sich nun die Frage: Warum der ganze Aufwand? Hätte es ein einfaches

```
gcc test.c -o test.exe
```

nicht auch getan?

Worauf ich antworte: Vom Prinzip her schon, nur wären dann die Vorgänge bei diesem Aufruf im Dunklen geblieben. Und dies hat 2 entscheidende Nachteile:

- Zum einen erzeugen der Präprozessor, der Compiler und der Linker jeweils eigene, unterschiedliche Fehler und Warnungen. Man kann mit diesen einfacher umgehen, wenn man weiß, in welcher Weise diese Vorgänge zusammenhängen.
- Zum anderen ist dieses Wissen die Grundlage zum Erstellen von Projekten, die sich über mehrere Files erstrecken.

Womit wir schon beim nächsten Thema wären.

Modularisierung von Programmen

Quellcode hat die Angewohnheit zu wachsen. Schnell zu wachsen. Innerhalb weniger Tage und Wochen kann aus dem überschaubaren Projekt von wenigen hundert Zeilen Code ein Monster mit ein paar tausend Zeilen Code werden. Dies macht den Code nicht nur unübersichtlich, sondert sorgt auch dafür, dass das Kompilieren länger dauert.

Wenn man jedoch das Programm in mehrere, von einander unabhängige Dateien steckt, dann muss man immer nur die kompilieren, die man geändert hat. Diese Ersparnis kann bei umfangreichen Projekten einen gewaltigen Unterschied ausmachen.

Nehmen wir an, unser Spiel bestehe aus diesen Komponenten:

- Spiel (`game.c`): Das eigentliche Spiel. Benötigt all die Resourcen, die das Hauptteil reserviert hat. Am Ende wird der Punktestand übermittelt, damit die Wertung in die Bestenliste übernommen werden kann.
- Menü und Hiscore (`menu.c`): Das Hauptmenü und die Hiscore-Verwaltung. Übermittelt dem Hauptteil den ausgewählten Menüpunkt und die Hiscore-Informationen.
- Hauptteil (`main.c`): Initialisiert Grafikmodus, Sound und Eingabe. Lädt die stets benötigten Grafiken, Schriftarten und Sounds. Sorgt bei Programmende dafür, dass auch jede dieser Ressourcen wieder freigegeben wird. Ruft sowohl das Menü als auch das eigentliche Spiel auf.

Es wird gleich deutlich, dass alle Teile auf Funktionen des Hauptteils zugreifen müssen. Die Module »Menü« und »Spiel« müssen nichts voneinander wissen (Spielestart und die Kommunikation der erzielten Punkte können über den Hauptteil gehen).

Die Schnittstellen der jeweiligen Komponenten werden in einer Header Datei definiert, die den gleichen Namen wie das jeweilige Modul hat. Das erlaubt es dem Compiler mit Funktionen und Variablen zu arbeiten, deren eigentliche Implementierung noch gar nicht feststeht. Dem Compiler reicht die Definition

```
int runGame(void);
```

um den benötigten Code für den Aufruf der Funktion und die Behandlung des Rückgabewertes zu erzeugen. Sobald dann beim Linken die Einzelheiten der Funktion bekannt sind, kann der Aufruf der eigentlichen Funktion erfolgen.

Vergisst man die Implementierung einer Funktion, oder (was häufiger vorkommt) verschreibt man sich im Funktionsnamen in der Implementierung, dann meldet (erst) der Linker einen Fehler.

Die gute Nachricht ist, dass wir auch alle Komponenten einzeln kompilieren, warten und ändern können. Dies kann vor allem bei größeren Projekten eine Menge Zeit sparen

```
gcc -c menu.c -o menu.o
gcc -c game.c -o game.o
gcc -c main.c -o main.o

gcc menu.o game.o main.o -o hyper.exe
```

Wenn sich nun im Menü etwas ändert, dann reicht es, diese eine Datei neu zu kompilieren und das Projekt neu zu linken.

```
gcc -c menu.c -o menu.o
gcc menu.o game.o main.o -o hyper.exe
```

Ein weiterer Vorteil kommt dann zu tragen, wenn mehrere Entwickler am gleichen Projekt arbeiten. Weist man jedem Entwickler eine Reihe von Source-Dateien zu, so kann dieser diese Files unabhängig von den anderen bearbeiten.

Die einzige Schwierigkeit besteht darin, die Abhängigkeiten im Kopf zu behalten und zu überprüfen, welche Dateien sich geändert haben. Da dies eine langweilige und wiederkehrende Tätigkeit ist, und die meisten Programmierer von Natur aus langweilige und wiederkehrende Aufgaben verabscheuen, ist es nur natürlich, dass jemand ein Tool geschrieben hat, um diese Aufgabe zu automatisieren.

Make

Schon im Jahre 1975 hat Stuart Feldmann ein Tool veröffentlicht, welches das »Bauen« von Software automatisiert: Make. Dieses Tool wird heute als eines der einflussreichsten und am häufig eingesetzten Programmierwerkzeuge weltweit angesehen.

Damit Make seine Arbeit verrichten kann, muss man eine Beschreibung der Abhängigkeiten des Projektes anlegen. Die einfachste Form eines solchen »Makefiles« könnte für unser Projekt in etwa so aussehen:

```
hyper.exe: main.o game.o menu.o
        gcc -o hyper.exe main.o game.o menu.o
main:o: main.c
        gcc -o main.o -c main.c
menu.o: menu.c
        gcc -o menu.o -c menu.c
game.o: game.c
    gcc -o game.o -c game.c
```

Am Anfang steht das Target (Ziel) der Regel, dann folgen die Dateien auf denen das Ziel basiert. Darauf folgen, mittels Tabulator eingerückt, die Befehle, die das Target erstellen. Dies ist auch die einzige wirkliche Schwierigkeit in einem Makefile. Rückt man die Befehle nicht mittels Tab ein (sondern zum Beispiel mittels Leerzeichen), dann meldet Make einen Fehler. Keine große Sache sollte man denken. Leider ersetzen einige Editoren Tabs ungefragt durch Leerzeichen. Aus diesem Grund sollte bei »unerklärlichen« Fehlern im Makefile erst mal überprüft werden, ob auch wirklich alle Einrückungen durch Tabs erfolgt sind.

Wie funktioniert Make?

Wenn Make aufgerufen wird, wird erst überprüft welches Target erzeugt werden muss. Wurde kein Target auf der Kommandozeile übergeben, dann wird das in der Datei an erster Stelle stehende Target benutzt.

In unserem Fall wäre dies das Target »`hyper.exe`«. Make überprüft nun ob es einen Grund gibt, diese Datei neu zu erstellen. Gründe wären zum Beispiel die Tatsache, dass »`hyper.exe`« noch gar nicht existiert, oder eine der Dateien, von denen es abhängig ist (`main.o`, `game.o` und `menu.o`) jünger sind als `hyper.exe`. »Jünger« bedeutet in diesem Zusammenhang, dass das Änderungsdatum der Datei neuer ist.

Außerdem überprüft Make ebenfalls, ob es einen Grund gibt, eine der Dateien `main.o`, `game.o` oder `menu.o` neu zu erstellen. Zu diesem Zweck werden die anderen angegebenen Regeln benutzt.

Und so hangelt sich Make von Regel zu Regel weiter und sorgt dafür, dass all die Komponenten neu erzeugt werden, bei denen sich etwas geändert hat.

Nun sind ja Programmierer, wie bereits vorher erwähnt, von Natur aus eher faul. Und sobald sich Dinge wiederholen, sehen Sie einen Grund dies zu optimieren.

Variablen

Variablen in Make funktionieren so in etwa so wie die bereits besprochenen Präprozessor Makros.

```
CC = gcc
OBJECTS = main.o menu.o game.o
hyper.exe: ${OBJECTS}

    ${CC} -o hyper.exe ${OBJECTS}
```

Die allgemeine Syntax ist also:

```
variable = wert
```

und zum Aufrufen:

```
${variable}
```

Neben den Variablen, die der User definieren kann, gibt es auch einige wichtige interne Bezeichner. So wird das aktuelle Target in der Variablen `$@` gespeichert.

Ich möchte die Gelegenheit nutzen, darauf hinzuweisen, dass die Verwendung von sprechenden Bezeichnernamen durchaus Sinn macht. Hätte sich der Programmierer von Make für `$TARGET` anstelle von `$@` entschieden, dann wären Makefiles heute deutlich lesbarer.

Aber da die Variable nun mal `$@` heißt, sieht nun die Regel für `hyper.exe` wie folgt aus:

```
hyper.exe : ${OBJECTS}
    ${CC} -o $@ ${OBJECTS}
```

Eine weitere wichtige Variable ist $<, die die erste Quelle aus der Liste der angegebenen Quelldateien enthält. Diese Variable ist sehr wichtig, wenn man seine Make-Regeln durch Muster noch weiter verallgemeinern will.

Muster-Regeln

Muster-Regeln sind Regeln für Dateien, deren Namen einem bestimmten Muster folgen. Dabei steht das Zeichen »%« für eine beliebige Kombination von Buchstaben und Zahlen. Eine allgemeine Regel für die Erstellung von .o aus .c Dateien könnte zum Beispiel so aussehen:

```
%.o: %.c
    ${CC} -c ${CFLAGS} -o $@ $<
```

Was in Klartext etwa soviel bedeutet wie »eine Datei mit der Endung .o kann aus einer Datei mit der Endung .c erzeugt werden, wenn der Teil vor der Endung gleich ist. Zum Erzeugen muss CC mit den Parametern -c und den in CFLAGS gespeicherten Flags aufgerufen werden«.

Wichtig ist, dass $@ in diesem Fall den tatsächlichen Namen der Ergebnisdatei enthält, und $< den tatsächlichen Namen der Quelldatei. CFLAGS ist ebenfalls eine Standardvariable, die Parameter für den Compiler enthält. So könnte CFLAGS während der Entwicklung auf

```
CFLAGS = -Wall -g
```

gestellt sein, damit alle Warnungen ausgegeben werden und in das entstehende File Debug-Informationen integriert werden. Für die endgültige Version kann dann CFLAGS abgeändert werden:

```
CFLAGS = -O3 -s
```

Was dafür sorgt, dass der Compiler den Sourcecode noch stärker auf Geschwindigkeit optimiert und nicht benötigte Symbole aus der Datei entfernt.

Geschafft. Dies waren all die benötigten Grundlagen, um mit den GNU Compiler Tools umgehen zu können. Für die meisten Aufgaben dürfte dieses Verständnis reichen. Bei den wenigen Gelegenheiten, in denen ein tieferes Wissen notwendig ist, werde ich noch mal darauf zurückkommen.

Zeiger

Zeiger werden in den meisten Einsteigerkursen nur oberflächlich behandelt. Man sieht auch häufiger Posts in den einschlägigen Entwicklerforen, aus denen deutlich hervorgeht, dass Pointer vielen Leuten suspekt sind.

Was ja eigentlich kein Problem wäre, wenn Zeiger und die damit verbundenen Techniken und Möglichkeiten nicht so grundlegend in C und C++ wären.

Zeiger sind keine komplizierte Sache – im Gegenteil: Sie sind eigentlich recht simpel. Wie der Name schon sagt, zeigen Pointer auf eine bestimmte Stelle im Hauptspeicher des Rechners. Wenn man es genau nimmt, dann sind Zeiger eigentlich nichts anderes als eine normale Integerzahl, die als Speicheradresse benutzt wird. Aus diesem Grund kann man mit Zeigern alles machen, was auch mit Integerwerten funktioniert.

In C kann man auch einen Zeiger wie ein `int` behandeln:

```
void *pointer;
int integer = 20;
pointer = integer;
```

Obwohl es sich hierbei um gültigen C Code handelt, kann es bei der Ausführung durchaus zu Problemen kommen (die Chance, dass sich die Speicheradresse 20 innerhalb eines gültigen Bereichs für das eigene Programm befindet, sind doch eher gering).

In C++ würde der obige Code nicht kompilieren. C++ hat eine strengere Typüberprüfung, und deswegen muss man explizit angeben, dass man einen Zeiger zu einem `int` umwandeln möchte und umgekehrt.

```
void *pointer;
int integer = 20;
pointer = (void*) integer;
```

Dies ist jedoch der einzige Unterschied. Sobald die Werte mal zugewiesen sind, verhalten sich C und C++ identisch.

*, & und ->

Mit ein Grund, warum Zeiger wohl meist etwas argwöhnisch betrachtet werden, dürften wohl die Operatoren sein, mit denen man auf Zeiger zu-

greift. Aber trotz ihres kryptischen Aussehens sind diese Operatoren doch eigentlich eher einfacher Natur.

Operator	Bedeutung
*foo	»Das, worauf foo zeigt«.
&foo	Die Adresse von foo
foo->bar	Entspricht (*foo).bar. Wird benutzt, um auf Mitglieder von Strukturen zuzugreifen, wenn nur ein Zeiger auf die Struktur gegeben ist. Dieser Operator hat also mehr oder weniger rein kosmetische Bedeutung.

Tabelle 7.1: Zeiger-Operatoren

Die Bedeutung wird klarer wenn man sich ein kleines Beispiel ansieht:

```
/* Das worauf "pointerToInt" zeigt ist ein int */
int *pointerToInt = NULL;
int value = 10;
/* "pointerToInt" wird auf die Adresse von "value" gesetzt */
pointerToInt = &value;
/* Das worauf "pointerToInt" zeigt (also "value") wird auf 4711 ge-
setzt */
*pointerToInt = 4711;
/* Ausgabe von value -> Ergebnis 4711
printf("value = %i\n", value);
```

Zeiger und Arrays

Nun könnte man annehmen, dass, wenn ein Zeiger nur ein `int` ist, man auch ganz normal mit Zeigern rechnen kann. Und zumindest bei reinen Zeigern (`void*`) und Zeigern auf Zeichen (`char*`) trifft das auch so zu. Bei Zeigern auf andere Datentypen sieht es jedoch etwas anders aus.

In C und C++ können Zeiger und Arrays beliebig ausgetauscht werden. Jeder Zeiger kann wie ein Array angesprochen werden, jedes Array wie ein Zeiger.

```
#include <stdio.h>
        int valueArray[] =
                                {10, 20, 30, 40, 50, 60, 70, 80, 90};
        int *pointer;
        int counter;
        pointer = valueArray;
        /* Zugriff auf einen Zeiger als ein Array */
        pointer[3] = 35;
        /* Zugriff auf ein Array als Zeiger */
        *valueArray = 15
        for (counter=0; counter < 9; counter++) {
            printf("valueArray[%i] = %i\n",
                    counter, valueArray[counter]);
        }
        return 0;
}
```

Wenn man dieses Programm kompiliert, erhält man folgende Ausgabe:

```
valueArray[0] = 15
valueArray[1] = 20
valueArray[2] = 30
valueArray[3] = 35
valueArray[4] = 50
valueArray[5] = 60
valueArray[6] = 70
valueArray[7] = 80
valueArray[8] = 90
```

Der erste Eintrag wurde durch

```
*valueArray = 15;
```

von 10 auf 15 erhöht. Der 4te Eintrag (`valueArray[3]`) wurde von 40 auf 35 erniedrigt.

Die Ähnlichkeit geht sogar so weit, dass, wenn man einen Zeiger erhöht oder erniedrigt, automatisch die Größe des Grunddatentyps in Betracht gezogen wird.

```
#include <stdio.h>

int main(int argc, char** argv) {
    int *zeiger;
    int array[10];
    int adresse = 0;
    int a;
```

```
    zeiger = array;
    /* Vergleiche array_[0] mit dem, was an der
       Adresse (zeiger+0) steht */
    if (array[0] == *(zeiger +0)) {
        printf("ok\n");
    }

    /* Vergleiche array_[1] mit dem, was an der
       Adresse (zeiger+1) steht */
    if (array[1] == *(zeiger +1)) {
        printf("ok\n");
    }
    if (&( array[2] ) ==  zeiger +2) {
        printf("ok\n");
    }
    /* Und nun mal etwas genauer */
    for (a=0; a < 10; a++) {
        adresse = (zeiger +a);
        printf("Adresse von zeiger+%i= %i    ",
               a, adresse);
        adresse = &(array[a]);
        printf("&(array[%i] = %i\n", a, adresse);
    }

    return 0;
}
```

Dieses Programm liefert folgende Ausgabe:

```
ok
ok
ok
Adresse von zeiger+0= 7601560    &(array[0] = 7601560
Adresse von zeiger+1= 7601564    &(array[1] = 7601564
Adresse von zeiger+2= 7601568    &(array[2] = 7601568
Adresse von zeiger+3= 7601572    &(array[3] = 7601572
Adresse von zeiger+4= 7601576    &(array[4] = 7601576
Adresse von zeiger+5= 7601580    &(array[5] = 7601580
Adresse von zeiger+6= 7601584    &(array[6] = 7601584
Adresse von zeiger+7= 7601588    &(array[7] = 7601588
Adresse von zeiger+8= 7601592    &(array[8] = 7601592
Adresse von zeiger+9= 7601596    &(array[9] = 7601596
```

Dies zeigt ganz klar, dass das Ergebnis von `array[a]` und `*(zeiger+a)` gleich ist. Es zeigt auch, dass ein `int` 4 Bytes Speicher belegt (die Abstände zwischen den einzelnen Adressen ist 4).

Wie kann man nun die einzelnen Bytes eines `int`s ansprechen? Zeigermagie macht es möglich:

```
/* Zahl in hexadezimal */
int zahl =0x11223344;
/* Typ "Zeiger auf int" in "Zeiger auf char" ändern */
char *zeiger = (char*) &zahl;
int a;
for (a=0; a < 4; a++) {
    printf("%x ", zeiger[a]);
}
printf("\n");
```

Das Ergebnis sieht auf einem Intel oder kompatiblen System so aus:

```
44 33 22 11
```

Diese Anordnung mag zwar überraschen, ist aber auf Intel Systemen der Standart. Andere CPU Hersteller, zum Beispiel Motorola, ziehen die umgekehrte Reihenfolge vor. Auf einem Motorola System sähe die Reihenfolge so aus:

```
11 22 33 44
```

Diese Unterschiede können manchmal Probleme machen, insbesondere wenn man ein Spiel für mehrere Plattformen entwickeln möchte, da man nun beim Speichern der Daten auf die Platte, bzw. beim Laden von der Platte byteweise vorgehen und dabei selbst dafür sorgen muss, dass die Bytes in der richtigen Reihenfolge im Speicher landen. Glücklicherweise enthält Allegro (die Bibliothek, die wir für die Erstellung plattformunabhängiger Spiele benutzen werden) bereits alle Funktionen, um Daten systemunabhängig speichern zu können.

C Strings

Bisher waren alle Beispiele doch eher weit hergeholt. Ein Bereich, in dem man jedoch eigentlich immer in der einen oder anderen Form mit Zeigern zu tun hat, sind Zeichenketten. In C sind Strings Arrays von `char`. Da Arrays wie Zeiger behandelt werden, hat man immer, wenn es um Zeichenketten geht mit Zeigern zu tun.

Ein weiterer wichtiger Punkt ist, dass das Ende eines Strings durch den Wert 0 repräsentiert wird (dem Wert 0, nicht dem Zeichen ‚0'). Man spricht deswegen auch von »Null terminierten Zeichenketten« (Engl.: zero terminated strings).

Alle C-Standard-String-Funktionen (und die Textausgabefunktionen von Allegro) erwarten Null terminierte Strings.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv) {
    /* 1kB Speicher */
    char text[1024];
    /* erstes Zeichen auf0 -> Leerstring */
    text[0] = '\0';
    printf("Laenge von >%s<: %i\n,
            text, strlen(text));
    text[0] = ,T';
    text[1] = ,e';
    text[2] = ,s';
    text[3] = ,t';
    text[4] = ,\0';
    printf("Laenge von >%s<: %i\n,
            text, strlen(text));
    return 0;
}
```

Die Länge des Strings hängt also nur von der Position der Null ab. Wenn nun keine Null explizit gesetzt ist, dann wird der gesamte Speicherbereich vom Anfang des Strings bis zur nächsten, zufällig im Speicher platzierten Null als String betrachtet. Dass dies normalerweise nicht beabsichtigt ist, ist klar. Darüber hinaus kann es zu Schutzverletzungen und damit zum Programmabsturz führen.

Bei Stringkonstanten, wie »Hallo!«, hängt der Compiler die abschließende Null automatisch an. Der String »Hallo!« würde dann also 6 Zeichen belegen: `H`, `a`, `l`, `l`, `o`, `!`, `\0`.

Hier ist eine Liste der häufiger benutzten String-Funktionen, inklusive einer kurzen Erklärung:

`int strlen(char* str)`: Gibt die Anzahl der Zeichen vor der abschließenden Null an. Der Rückgabewert von `strlen("Hallo!")` ist also 5.

`char strdup(char* str)`: Erstellt eine Kopie von `str`. Der Speicher für die Kopie wird dynamisch angelegt, muss also explizit freigegeben werden. Mehr zum dynamischen Bereitstellen von Speicher in einem Augenblick.

`strncpy(char *ziel, char*quelle, int anzahl)`: Kopiert maximal `anzahl` Zeichen von `quelle` nach `ziel`.

`int strncmp(char* string1, char* string2)`: Vergleicht `string1` mit `string2`. Wenn `string1` < `string2` (zum Beispiel wenn `string1` »AAA« ist, und `string2` »BBB«), wird –1 zurückgegeben. Ist `string1` größer als `string2`, wird +1 zurückgegeben. Sind die beiden Zeichenketten identisch, dann wird 0 zurückgegeben.

Speicher anlegen und freigeben

Der Hauptvorteil von Zeigern ist, dass er auf dynamisch zugewiesenen Speicher zeigen kann. Dynamischer Speicher ist immer dann wichtig, wenn zum Zeitpunkt der Kompilierung nicht feststeht, wieviel Speicher nun genau benötigt wird. Ein einfaches Beispiel: Für eine Umsetzung von »Schiffe versenken« wollen wir es dem Spieler erlauben eigene Spielpläne anzulegen. Die Größe dieser Karten ist frei wählbar, ebenso die Lage möglicher Inseln. Die Größe des benötigten Speichers ist Breite * Höhe der Karte. Nun kennen wir aber beides noch nicht zur Kompilezeit, sondern erst, wenn der Level geladen wird. In der Standard-C-Bibliothek gibt es eine Funktion `malloc()` (Engl. für Memory Allocation, also: Speicher Bereitstellung), die uns einen Zeiger auf einen Bereich der entsprechenden Größe zurückgibt (sofern genug Speicher vorhanden ist).

```
/* Die Map Struktur */
typedef struct {
    int width;
    int height;
    char *data;
} Map;

Map* createMap(int width, int height) {
    /* Zuerst brauchen wir einen Zeiger auf die Karten Struktur */
    Map *map = (Map*) malloc( sizeof(Map) );
    if (map == NULL) {
        /* Kein Speicher mehr frei */
        return NULL;
    }
    /* Eintragen Breite und Höhe… */
    map->width = width;
    map->height = height;
    /* und dann Speicher für die eigentliche Karte */
```

```
    map->data = (char*) malloc(width*height);
    if (map->data == NULL) {
        /* Kein Speicher für die Level daten
           Also geben wir den Speicher für die Karte
           Wieder frei, und geben NULL zurück
         */
        free(map);
        return NULL;
    }
    return map;
}
```

Die Map-Struktur enthält alle Informationen, die wir brauchen, um mit der Karte zu arbeiten: Höhe, Breite und die eigentlichen Daten. Wenn wir nun eine Karte dynamisch erzeugen wollen, dann brauchen wir zuerst einmal Speicher für die Map Struktur.

```
Map *map = (Map*) malloc( sizeof(Map) );
```

Der `sizeof()` Operator liefert die Größe eines Datentyps oder einer Struktur zurück. Diese Anweisung würde also im Klartext bedeuten: »Reserviere die Menge an Speicher, die eine `Map` Struktur benötigt. Wandele den Zeiger auf diesen Bereich in einen Zeiger auf eine `Map` Struktur um und weise ihn der Variablen `map` zu.«

Da `malloc()` auch `NULL` zurückgeben kann, ist es wichtig den Rückgabewert zu überprüfen. Ist der Rückgabewert nämlich `NULL`, so konnte kein Speicher der angeforderten Größe angelegt werden.

Wenn der Speicher erfolgreich angelegt wurde, dann werden die `width` und `height` Komponenten der Struktur auf die übergebenen Werte gesetzt. Dann wird der eigentliche Speicher für den Level alloziert.

```
map->data = (char*) malloc(width*height);
```

Auch hier wird der Rückgabewert von `malloc()` auf den erforderlichen Typ gecastet.

Wenn `malloc()` hierbei `NULL` zurückliefert müssen wir erst den bereits erfolgreich angelegten Speicher für die `Map` Struktur freigeben. Würden wir dies vergessen, dann hätten wir ein Speicherleck, das bedeutet, dass dieser Speicher von dem Programm weder freigegeben noch benutzt werden kann. Er wäre also quasi verloren.

Wenn `malloc()` jedoch erfolgreich verläuft, dann geben wir `map` zurück und die Funktion ist erfolgreich beendet worden.

Speicher, der mit `malloc()` angelegt wurde, wird durch einen Aufruf der Funktion `free()` wieder freigegeben. Eine Funktion, die den von `createMap()` angelegten Speicher wieder freigibt, könnte so aussehen:

```
void freeMap(Map* map) {
    if (map != NULL) {
        if (map->data != NULL) {
            free(map->data);
        }
        free(map);
    }
}
```

Ich kann nur empfehlen, für jedes `malloc()` auch sofort ein entsprechendes `free()` zu schreiben, um mögliche Speicherlecks auszuschließen.

8 Von C zu C++

Dieses Kapitel bietet eine Zusammenfassung der wichtigsten Unterschiede von C und C++. Bisher waren alle Beispiele in C. Dies hat einen einfachen Grund: Die Bibliothek, die uns die Plattform-übergreifenden Routinen für Musik, Grafik und Ein-/Ausgabe bereitstellt, ist in C geschrieben.

Aus diesem Grund werden wir häufiger mit C-Routinen arbeiten. Auch sind die mit Allegro kommenden Beispiele alle in C geschrieben. Es macht also durchaus Sinn, sich auch als eingefleischter C++-Programmierer mit den Eigenheiten von C auszukennen.

In diesem Buch werde ich eine Mischung aus prozeduraler und ***O**bjektorientierter **P**rogrammierung* (OOP) verwenden.

Dies hat einige Vorteile: Wir können die C++-Standard-Klassen benutzen, können Methoden überladen, Default-Parameter verwenden und vieles mehr. Wir behalten aber weiterhin die Flexibilität des prozeduralen Ansatzes. Nicht alles muss unbedingt ein gekapseltes Objekt sein. Auch benutzt Allegro an einigen Stellen Callback-Funktionen, die sich innerhalb von Klassen nur als statische Methode realisieren lassen würden.

In den nächsten Abschnitten werde ich erst auf die nicht-objektorientierten Eigenschaften von C++ eingehen und dann einen kurzen Einblick in die OOP-Features von C++ geben.

Funktionen in C++

Es gibt ein paar kleine, aber feine Erweiterungen Funktionen betreffend in C++. Angenommen, wir haben eine Funktion, die den Punktestand des Spielers erhöht. Wenn Sie aufgerufen wird, spielt sie ein Geräusch ab und erhöht dann die Punkte. Sie überprüft auch, ob die Punktzahl ausreicht, um ein Extraleben zu erreichen und wenn das der Fall ist, dann wird auch gleich dieses Leben auf addiert. Diese Funktion könnte wie folgt aussehen:

```
void incScore(int value) {
    int pointsNeededForExtraLife =
```

```
        (score / EXTRA_LIFE_POINTS)+1
        * EXTRA_LIFE_POINTS;
    if (score + value >= pointsNeededForExtraLife) {
        lifes++;
        play_sound(EXTRA_LIFE);
    }
    score += value;
    play_sound(BONUS);
}
```

Jetzt fügen wir später in der Entwicklung aber noch ein paar spezielle Gegenstände hinzu, bei denen ein anderer Sound abgespielt werden soll. Aus diesem Zweck ändern wir die Funktion ab, damit wir auch noch die Art des Sounds übergeben können:

```
void incScore(int value, int bonusType) {
    int pointsNeededForExtraLife =
        (score / EXTRA_LIFE_POINTS)+1
        * EXTRA_LIFE_POINTS;
    if (score + value >= pointsNeededForExtraLife) {
        lifes++;
        play_sound(EXTRA_LIFE);
    }
    score += value;
    play_sound(bonus_sounds[bonusType]);
}
```

Nun wird die Funktion aber an mehreren Stellen des Spiels aufgerufen, und wir suchen eine effektivere Möglichkeit des Problems Herr zu werden als jeden Aufruf per Hand zu ändern. Glücklicherweise erlaubt uns C++ Parametern einen Defaultwert mitzugeben. Dieser Defaultwert wird immer benutzt, wenn der Parameter nicht angegeben wird. Wenn wir die Funktion abändern, damit der Parameter `bonusType` den Defaultwert 0 erhält, dann sieht die Funktion so aus:

```
void incScore(int value, int bonusType = 0) {
    // Der Rest bleibt wie bisher
    // (Und man beachte diesen C++ Kommentar :-)
}
```

Wird beim Aufruf der Funktion der Parameter `bonusType` nicht angegeben, dann wird dies wie ein Aufruf behandelt, in dem als zweiter Parameter 0 übergeben wird. Defaultwerte sind immer dann praktisch, wenn ein Parameter die meiste Zeit über den gleichen Wert hat, aber in Ausnahmefällen auch einen anderen Wert haben könnte.

Eine Funktion kann beliebig viele Parameter mit Defaultwert haben. Allerdings können auf einen oder mehrere Parameter mit Defaultwert keine Parameter ohne Defaultwert folgen.

```
// Diese Funktion ist ok:
void example(int intParam = 0, char *textParam = "test", float
floatParam = 0.0) {
}

// Diese hier ist falsch, textParam hat keinen Defaultwert
void badExample(int intParam = 0, char *textParam, float floatParam =
0.0) {
}
```

Neben Defaultparametern gibt es noch eine weitere C++ Funktionalität, die mehr als praktisch ist: Das Überladen von Funktionen. Das erlaubt es uns, eine Funktion mit dem gleichen Namen, aber unterschiedlichen Aufruflisten zu benutzen.

Nehmen wir an, wir haben eine Funktion, welche die Darstellung einer Nachricht übernimmt. Ein Art Statusmeldung oder Messagebox. Die Nachricht, die angezeigt werden soll, kann entweder über eine eindeutige Identifikationsnummer angegeben werden (für Standard Meldungen) oder aus dem auszugebenen Text bestehen. In einigen wenigen Situationen könnte es auch Sinn machen, nur ein Bild anzuzeigen. In C müssten wir dafür 3 verschiedene Funktionen mit unterschiedlichen Namen schreiben. In C++ können wir einfach 3 Funktionen mit dem gleichen Namen, aber unterschiedlichen Parameterlisten erstellen.

```
void showMessage(int msgID);
void showMessage(char *text);
void showMessage(BITMAP *image);
```

Im eigentlichen Programm würden wir dann `showMessage()` mit einem gültigen Parameter aufrufen, und der Compiler übernimmt dann die Arbeit die korrekte Funktion aufzurufen.

Allerdings gibt es bei diesem Beispiel einen Fallstrick. Wir haben `showMessage()` sowohl mit einem `int` als auch mit Zeigertypen (`char*`, `BITMAP*`) überladen. Das Problem erhebt dann sein unschönes Haupt, wenn wir die Funktion wie folgt aufrufen:

```
showMessage(NULL);
```

Welche Funktion wird jetzt aufgerufen?

Wie wir wissen, sind Zeiger nichts anderes als Integerwerte. Und `NULL` ist zumeist einfach als 0 definiert. Für den Compiler ist es also klar, dass auf jeden Fall `showMessage(int msgID)` aufgerufen werden muss. In unserem Fall ist dies kein großes Problem, da es keinen Sinn macht `showMessage()` mit einem `NULL` Pointer aufzurufen. Bei anderen Funktionen könnte die Sache jedoch anders aussehen.

Wenn also eine Funktion sowohl mit einem `int` als auch einem Zeigerwert aufgerufen werden kann, ist Vorsicht geboten.

Überladen oder Defaultparameter?

Sowohl Defaultparameter als auch das Überladen von Funktionen erlauben es uns, eine Funktion mit unterschiedlichen Parameterlisten zu erstellen. Ob man nun in einem bestimmten Fall Defaultwerte oder doch lieber das Überladen benutzen soll, hängt von zwei von Dingen ab:

- ✔ Gibt es einen passenden Defaultwert für den Parameter?
- ✔ Wird für alle Parameter Optionen der gleiche Algorithmus verwendet?

Wenn es einen passenden Standartwert für den oder die Parameter gibt und für alle Parameter der gleiche Algorithmus verwendet wird, dann ist es in der Regal ratsam, Defaultparametern den Vorzug zu geben.

Wenn es unterschiedliche Algorithmen sind oder es schlicht und ergreifend keinen passenden Defaultwert gibt, dann muss man die Funktion überladen.

Strukturen in C++

Strukturen sind in C++ vom Prinzip her das Gleiche wie Klassen. So können Strukturen zum Beispiel auch Funktionen enthalten. Allerdings ist die Standardsichtbarkeit aller Komponenten einer `struct` in C++ `public`. Dadurch kann man Strukturen in C++ genauso wie in C benutzen.

```
typedef struct {
    int x, y;
    int w, h ;
} Rect;
Rect rect;
```

Ist in C++ ebenso gültig wie in C. Jedoch kann man sich in C++ etwas Schreibarbeit sparen, und das `typedef` weglassen. Der folgende Code ist die C++ Entsprechung des oben abgebildeten C Codes.

```
struct Rect{
    int x, y;
    int w, h ;
};
Rect rect;
```

Die Frage, wann (und ob) Strukturen in C++ verwendet werden sollen, erhitzt die Gemüter. Von einem puren C++ Blickwinkel sollte jedes Datenobjekt in einer Klasse gespeichert werden und Zugriff auf die Elemente nur über die entsprechenden Methoden erfolgen. Und aus Sicht der Kapselung ist dies auch der richtige Weg. Die Gegenargumente sind meist der Geschwindigkeitsverlust und die zusätzliche Komplexität.

Meiner Meinung nach sollten Sie diese Diskussionen denen überlassen, die gerne diskutieren, und sich von Fall für Fall für das eine oder andere entscheiden. Ich benutze normalerweise Strukturen, wenn ich Daten zusammenfassen möchte, diese aber nur selten im Ganzen anspreche oder die der direkte Zugriff auf die Datenelemente keine negativen Auswirkungen haben kann.

Mehr zum Thema Strukturen und Klassen folgt in einem Augenblick.

new und delete

In C++ haben `malloc()` und `free()` ausgedient. Speicher wird mittels `new` bereitgestellt und mit `delete` wieder freigegeben. Der Grund hierfür ist, dass Objekte in C++ über Konstruktoren erzeugt und durch Destruktoren wieder freigegeben werden müssen. Da es in C keine Objekte gibt, ruft `malloc()` natürlich auch nicht den entsprechenden Konstruktor auf. Aus diesem Grund kann man mittels `malloc()` keine C++ Objekte dynamisch erzeugen. Nun könnte man natürlich für primitive Datentypen wie `int`, `char`, `float` usw. weiterhin `malloc()` nutzen, aber davon würde ich abraten. Wird ein Speicherblock auf die falsche Art und Weise freigegeben, so ist das Ergebnis nicht definiert. Von Speicherlecks bis hin zu Programmabstürzen kann alles passieren. Es ist also sicherer, wenn Sie versuchen, sich auf eine Methode des Speicher-Anlegens zu beschränken.

Und new und delete sind mindestens genauso einfach (wenn nicht noch einfacher) zu handhaben wie die entsprechenden C Funktionen.

```
struct Rect{
    int x, y;
    int w, h;
};
int *intPointer = new int;
char *charArrayPointer = new char[256];
Rect *rectangle = new Rect;
```

Der Speicher wird einfach durch einen Aufruf von new <Datentyp> angelegt. Sehr viel einfacher geht es beinahe nicht mehr. Das Freigeben des Speichers ist genauso einfach:

```
delete intPointer;
delete [] charArrayPointer;
delete rectangle;
```

Die einzige Überraschung hier ist eventuell die Freigabe des Arrays. Die Angabe der Klammern nach delete ist notwendig, da anhand des übergebenen Zeigers nicht entschieden werden kann, ob nun ein einzelnes Element oder ein Array von Elementen freigegeben werden muss. Vergisst man bei der Freigabe die [], so wird nur das erste Element des Arrays freigegeben. Ein Speicherleck wäre die Folge.

Objektorientierte Programmierung

C++ erlaubt es, objektorientiert zu programmieren. Objekte, also »Dinge«, werden mittels Klassen im Programm abgebildet. Eine Klasse ist eine Verbindung aus Daten und der für den Zugriff auf diese Daten notwendigen Methoden. Dies klingt im ersten Augenblick kompliziert, ist aber recht einfach. In C++ ist eine Klasse im Grunde nichts weiter als eine C Struktur die auch Funktionen enthalten kann. Zusätzlich kann der Zugriff auf die Daten und Methoden der Funktion eingeschränkt werden.

Die nun folgende Version der Klasse Point hat noch ein paar Schwachstellen, die wir im Laufe dieses Kapitels jedoch noch ausmerzen werden.

```
class Point {
public:
    int x, y;
```

```
    Point() {
        x = y = 0;
    }
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }
    ~Point() {
    }
    void setPosition(int x, int y) {
        this->x = x;
        this->y = y;
    }
    void move(int dx, int dy) {
        x += dx;
        y += dy;
    }
};
```

Diese Klasse enthält neben den Daten auch zwei Konstruktoren, einen Destruktor und zwei Methoden. Aber gehen wir doch einfach mal von oben nach unten alles durch:

```
class Point {
public:
```

Die erste Zeile startet die Klassendefinition und ist analog zu der Art und Weise wie in C++ Strukturen definiert werden. Das `public:` in der nächsten Zeile jedoch ist neu. Es regelt die Art des Zugriffs auf die folgenden Datenkomponenten und Methoden der Klasse. Der `public` (»öffentlich«) Zugriff bedeutet, dass die folgenden Methoden und Variablen für jeden zugänglich sind. Neben `public` gibt es noch `protected` (»geschützt«) und `private` (»privat«).

Bei der Variablendeklaration gibt es nichts Neues, also überspringen wir diese und gehen direkt zu den beiden Konstruktoren.

```
    Point() {
        x = y 0;
    }
    Point (int x, int y) {
        this->x = x;
        this->y = y;
    }
```

Ein Konstruktor ist eine besondere Funktion, die bei der Erstellung des Objekts aufgerufen wird und keinen Rückgabewert hat. Den Konstruktor

ohne Parameter nennt man den Standardkonstruktor (Default constructor). Er wird immer dann aufgerufen, wenn ein Objekt ohne weitere Angaben erzeugt wird. Dies ist zum Beispiel beim Anlegen eines Arrays der Fall.

```
// Aufruf des Standardkonstruktors
Point point;
Point *pointPointer = new Point();

// Aufruf des Standardkonstruktors für
// alle 200 Elemente
Point pointArray[200];
```

Alles was der Standardkonstruktor in unserem Beispiel macht, ist die x- und y- Komponenten auf 0 zu setzen.

Der zweite Konstruktor nimmt 2 Parameter. Dies erlaubt es uns dem Punkt direkt bei der Erstellung gültige Werte zu zuweisen.

```
Point point(200, 100);
Point pointPointer = new Point(200, 100);
```

In unserem Fall nimmt der Konstruktor 2 Parameter und setzt damit die klasseneigenen `x` und `y` Variablen. Der `this` Zeiger, der in diesem Konstruktor benutzt wird, ist in jeder Klasse definiert und zeigt auf die aktuelle Instanz, also quasi auf das Objekt selbst. Der Grund, warum er hier eingesetzt wird ist einfach der, dass es eine Überschneidung der Namensgebung zwischen den Parametern des Konstruktors und den Komponenten der Klasse gibt.

```
~Point() {

}
```

Der Destruktor hat ebenso wie der Konstruktor keinen Rückgabewert. Er kann auch keine Parameter haben. Er muss immer mit einer Tilde ~ beginnen. Für jede Klasse kann es nur einen Destruktor geben. In diesem Beispiel ist der Destruktor-Körper leer. Wenn die Klasse jedoch Speicher dynamisch angelegt hätte, dann wäre der Destruktor die richtige Stelle, den Speicher wieder freizugeben.

In der `setPosition()` Methode gibt es nicht viel zu sehen. Auch hier wird der `this` Pointer wieder eingesetzt, um auf die Komponenten zuzugreifen, die durch die Parameter der Methode verdeckt wurden. In der `move()` Methode ist dies nicht notwendig.

Hat-Ein und Ist-Ein

Ein Objekt für sich genommen ist eine eher langweilige Angelegenheit. Doch wenn man es mit anderen Objekten in Beziehung setzt, ändert sich dies schlagartig.

Die am häufigsten verwendeten Beziehungen sind die Hat-Ein- und Ist-Ein-Beziehungen. Ein Cabrio **ist ein** Auto. Aber nicht jedes Auto ist ein Cabrio. Ist doch eine einfache und einleuchtende Sache, oder? Wenn Sie diese Frage mit »Ja« beantwortet haben, dann haben Sie schon eine der größeren Hürden im OO-Verständnis genommen.

Wenn Sie eine Klasse von einer anderen Klasse ableiten, dann ist die abgeleitete Klasse immer auch ein Typ der Basis Klasse.

Klingt nicht mehr ganz so verständlich wie das erste Beispiel, aber wenn Sie diesen Satz leicht abändern, dann entspricht er in etwa: »Wenn Sie aus einem Auto ein Cabrio machen, dann ist es immer noch ein Auto«.

Die Hat-Ein-Beziehung ist noch eine Stufe einfacher. Ein Auto hat Räder. Nicht jedes Rad gehört unbedingt zu einem Auto.

Auch dies klingt doch durchaus einleuchtend. Und mit diesen beiden Beziehungen werden Sie in der Regel jede Beziehung zwischen Objekten beschreiben können.

Eine Ist-Ein-Beziehung wird über eine Ableitung implementiert, die Hat-Ein-Beziehung durch das Anlegen von Variablen innerhalb der Klasse.

Angenommen, Sie haben eine Klasse, die einen Punkt repräsentiert, und wollen nun eine Linien-Klasse erstellen. Die Linie und der Punkt haben sowohl von ihren Daten als auch von den Methoden einiges gemeinsam. Nur braucht man für die Linie ein Paar von Koordinaten mehr.

Die Frage, ob man nun die Linie vom Punkt ableiten soll, kann man nun recht einfach klären. Welcher dieser Sätze klingt eher nach unserem normalen Verständnis einer Line?

- ✔ Eine Linie ist ein Punkt, der mit einem zweiten Punkt verbunden ist.
- ✔ Eine Line hat zwei Punkte, die Ihre Lage im Raum angeben.

Ist eine Line ein Punkt? Oder hat eine Linie zwei Punkte?

Wenn Sie sich für letzteres entschieden haben, dann sind wir einer Meinung. Wenn Sie denken, dass eine Linie ein Punkt ist, dann sollten Sie kurz über die Konsequenzen dieser Entscheidung nachdenken.

Wenn Sie die Linie von der Punkt Klasse ableiten, dann können Sie eine Linie überall da übergeben, wo normalerweise ein Punkt gebraucht wird (da dann jede Linie auch ein Punkt ist).

So könnten Sie eine Linie als Position eines Objektes benutzen. Wenn Ihnen das seltsam vorkommt, dann sollten Sie evtl. noch einmal Ihre Entscheidung überdenken.

Ein anderes Beispiel: Bisher ist der Punkt 2 dimensional. Was, wenn wir einen 3 dimensionalen Punkt beschreiben möchten? Um die benötigte Beziehung zu ermitteln können wir wieder die Ergebnisse ausformulieren:

- ✔ Ein 3D-Punkt ist ein Punkt mit einer zusätzlichen Koordinate für die Z-Achse
- ✔ Ein 3D-Punkt hat einen 2D-Punkt und eine zusätzliche Z-Koordinate.

In diesem Fall ist die Lösung recht einfach, da ein Punkt ein Punkt bleibt, egal ob man seine Position im Raum oder der Fläche beschreibt.

Und ja, diese Beispiele waren eher einfach gestrickt. An der Methodik ändert sich jedoch nichts, wenn Sie versuchen, die Beziehung zwischen einem Level, der Spielerfigur, den Gegnern und dem Ausgang des Levels festzustellen.

Polymorphie

Die Idee hinter Polymorphie (oder zu Deutsch: Vielgestaltigkeit) ist recht simpel. Ein Objekt bleibt, was es ist. Aber wenn es so einfach ist, warum braucht man dann einen komplizierten Begriff wie Polymorphie, um diesen Sachverhalt auszudrücken?

Angenommen wir haben zwei Klassen, nennen wir sie mal »A« und »B«.

```
#include <iostream>

using namespace std;

class A {
```

```
public:
    void foo() {
        cout << "A-foo:";
        bar();
    }

    void bar() {
        cout << "A-bar" << endl;
    }

};

class B : public A {
public:

    void bar() {
        cout << "B-bar" << endl;
    }

};

int main(int, char**) {

    A a;
    a.foo();
}
```

Die Klasse a hat zwei Methoden, `foo()` und `bar()`, welche den Namen der Klasse und den Namen der Methode ausgeben. Die Methode `foo()` ruft, nachdem sie ihre Ausgabe getätigt hat, die Methode `bar()` auf. Wenn wir das obige Programm kompilieren und starten, bekommen wir auch die erwartete Ausgabe:

```
A-foo:A-bar
```

Was sollte passieren, wenn wir uns eine Instanz von `B` bauen, und dann von dieser `foo()` aufrufen? Da `B` keine überladene `foo()` Methode hat, wird die der Basisklasse aufgerufen. Innerhalb dieser wird dann `bar()` aufgerufen, welches überladen wurde. Folglich sollten wir also folgendes Ergebnis bekommen:

```
A-foo:B-bar
```

Also, ändern Sie die Methode um, und testen Sie es.

```
int main(int, char**) {
    B b;
    b.foo();
}
```

Ihre Ausgabe sollte nun so aussehen:

```
A-foo:A-bar
```

Es scheint total ignoriert zu werden, dass Sie die `bar()`-Methode der Klasse B überladen haben. Wie kann das sein?

Beim Kompilieren der Klasse `A` hat der Compiler keine Ahnung, dass Sie später einmal vorhaben die Klasse `B` von `A` abzuleiten. Er hat auch keine Ahnung, dass Sie vorhaben, die Methode `bar()` zu überladen. Aus diesem Grund wird der Aufruf von `bar()` innerhalb der Klasse `A` als ein Aufruf der `bar()` Methode von `A` umgesetzt.

Wir müssen also dem Compiler mitteilen, dass eine Funktion später eventuell überladen werden könnte. Sobald er das weiß, erzeugt er keinen direkten Verweis mehr auf die Funktion, sondern sieht erst in einer Tabelle nach, welche tatsächliche Funktion er denn nun aufrufen muss. Das entsprechende Schlüsselwort heißt »`virtual`«. Die Tabelle, welche die virtuellen Funktionen enthält, wird auch als *virtual function table*, oder kurz *VTable* bezeichnet.

Sobald wir die `bar()` Methode als `virtual` kennzeichnen, bekommen wir auch unser gewünschtes Ergebnis.

```
class A {
public:
    void foo() {
        cout << "A-foo:";
        bar();
    }

    virtual void bar() {
        cout << "A-bar" << endl;
    }

};
```

Der Aufruf von `b.foo()` erzeugt nun folgendes Ergebnis:

```
A-foo:B:bar
```

In einigen OO-Sprachen ist jede Methode standardmäßig »virtual.« Die Entwickler von C++ haben sich jedoch dafür entschieden es dem Programmierer selbst entscheiden zu lassen, ob er den mit Nachschlagen in der Tabelle verbundenen Overhead in Kauf nehmen möchte.

Zurück zum Thema Polymorphismus. Der Name kommt eigentlich daher, dass man jede abgeleitete Klasse da einsetzen kann, wo man die Basisklasse auch benutzen kann.

Wir können also die Funktion

```
void test(A a) {
    a.foo();
}
```

problemlos mit einer Variable vom Typ `B` aufrufen. Nur wird uns hier wieder eine böse Überraschung erwarten. Denn die Ausgabe der `test()` Funktion ist:

```
A-foo:A-bar
```

Es wird also offensichtlich nicht die `bar()` Methode der `B` Klasse aufgerufen. Was passiert hier?

In C++ werden ebenso wie in C die Parameter via Wert übergeben. Das heißt, die Funktion `test()` legt eine temporäre Variable vom Typ `A` an, und kopiert die Daten des übergebenen Parameters in diese neue, temporäre Instanz. Das Ergebnis ist dann eine Variable vom Type `A`. Diese hat natürlich auch keine überladene `bar()` Methode, also wird `A:bar` ausgegeben.

Wenn wir verhindern, dass eine neue Variable angelegt wird, und statt dessen mit der übergebenen Variable arbeiten, löst sich dieses Problem von selbst. Dazu haben wir 2 Möglichkeiten. Zum einen können wir einen Zeiger übergeben:

```
void test(A *a) {
    a->foo();
}
```

Zum anderen können wir ebenso eine Referenz auf das Objekt übergeben:

```
void test(A &a) {
    a.foo();
}
```

In beiden Fällen wird das übergebene Objekt direkt benutzt, und damit bleibt auch der Typ erhalten. Wenn Ihnen das alles etwas kompliziert erscheint, dann können Sie es auch auf folgende zwei Regeln reduzieren:

- Methoden, die in abgeleiteten Klassen überladen werden sollen, müssen als virtual deklariert werden.
- Übergeben Sie Objekte wann immer möglich als Zeiger oder als Referenz.

C++ Gotchas

In C++ gibt es einige häufig vorkommende Probleme. Ich werde hier auf einige der häufiger auftretenden Probleme eingehen.

Rückgabewerte prüfen

Wenn Sie versuchen neuen Speicher mittels `new` zu allozieren, dann kann es durchaus sein, dass dieser Versuch misslingt. In einem solchen Fall liefert Ihnen `new` den Wert `NULL` zurück.

Zwar ist Ihnen sicher bewusst, dass Sie jedes `new` checken sollten, aber in den meisten Fällen wird es doch gerne vergessen. Zum einen liegt dies daran, dass es einfach umständlich ist, jedes `new` zu überprüfen. In Verbindung mit dem »Prinzip Hoffnung« (»ich brauch ja nur 1024 Byte... das wird schon gut gehen«.) kann sich dies zu einer ziemlich explosiven Mischung entwickeln.

Ein weiteres Argument, das immer wieder auftaucht lautet in etwa so: »Wenn diese Speicheranfrage fehl schlägt, dann tut eh nichts mehr. Dann kann es ruhig abstürzen«.

Zum Glück gibt es jedoch eine Methode, um sich sowohl die `if` Abfragen um jedes `new`, als auch den Absturz zu ersparen. Es ist möglich eine Fehlerbehandlungsroutine zu setzen, die immer dann aufgerufen wird, sobald `new` nicht mehr in der Lage ist Speicher bereitzustellen.

```
void outOfMemHandler() {
    cerr << "Nicht genug Speicher" << endl;
    exit(0);
}
// und im Hauptprogramm
set_new_handler(outOfMemHandler);
```

Die Funktion `set_new_handler()` nimmt als Parameter eine Funktion, die keine Parameter erwartet und auch keinen Wert zurückgibt.

Der Unterschied zwischen einem plötzlichen Programmabbruch mit einer Fehlermeldung und einem Absturz (unter Unix eventuell mit einem Core-Dump, unter Windows mit einer »allgemeinen Schutzverletzung«) ist zwar nur gering, aber doch noch etwas aussagekräftiger. Und der Benutzer Ihres Programms wird es Ihnen auf jeden Fall danken.

Der langen Rede kurzer Sinn: Prüfen Sie den Rückgabewert von `new` und den anderen Methoden, die Speicher bereitstellen. Für die Fälle, in denen es keine Alternative außer einem überstürzten Programmende gibt, kann die `set_new_handler()` Methode weiterhelfen. Sie haben also ab jetzt keine Ausrede mehr, sich nicht um diese Problematik zu kümmern.

Geben Sie angelegten Speicher auch wieder frei

Dies ist eine sehr einfache Regel, und dennoch wird sie immer wieder gerne vergessen. Machen Sie es sich zur Angewohnheit, zu jedem `new` auch ein passendes `delete` zu schreiben.

Wenn Sie in einem Konstructor Speicher anfordern, dann schreiben Sie gleich die entsprechende Speicherfreigabe in den Destruktor.

Überprüfen Sie den Parameter des Zuweisungsoperators und geben Sie *this zurück

Wenn Sie einen Zuweisungsoperator für Ihre Klasse schreiben (und wie Sie gleich sehen werden, sollten Sie dies in den meisten Fällen tun), stellen Sie sicher, dass Sie auf Zuweisung an sich selbst überprüfen.

```
MyClass a;
a = a;
```

Eine solche Zuweisung mag zwar nicht sehr viel Sinn machen, ist aber durchaus möglich, insbesondere wenn Sie mit Referenzen und Zeigern arbeiten.

```
void foo(MyClass &a, MyClass& b) {
    a = b;
}
```

Können Sie in einem solchen Fall ausschließen, dass `a` und `b` unterschiedliche Namen für das gleiche Objekt sind? Insbesondere wenn diese Methode indirekt (also von anderen Methoden) aufgerufen wird?

Wenn Sie in Ihrem operator= eine if Abfrage hinzufügen, dann sind Sie gegen solche Probleme gewappnet.

```
MyClass& MyClass::operator=(const MyClass &rhs) {
    if (&rhs == this) {
        return *this;
    }
    // ...
   return *this;
}
```

Wie Sie sehen, gibt der operator= eine Referenz auf this zurück. Der Grund hierfür liegt in der Tatsache begründet, dass Sie in C++ Zuweisungsketten benutzen können:

```
a =b = c = d = f;
```

Wenn Sie dies in eine Funktionsschreibweise umwandeln, sieht das Ergebnis wie folgt aus:

```
a.operator=(b.operator=(c.operator=(d.operator=(f))));
```

Der Rückgabewert des operator= muss also auch als Parameter für den operator= benutzt werden können. Und da der Parameter nun mal eine Referenz auf die Klasse ist, muss auch der Rückgabewert eine Referenz auf die Klasse sein.

Schreiben Sie einen Copy Constructor und einen Zuweisungsoperator

Wenn Ihre Klasse Speicher dynamisch erzeugt (und hoffentlich auch wieder freigibt) dann sollten Sie sowohl einen Copy Constructor als auch einen Zuweisungsoperator zur Klasse hinzufügen.

Der Grund hierfür ist ebenso einfach wie erschreckend: Wenn Sie es nicht tun, erzeugt der Compiler diese beiden Methoden für Sie. Allerdings hat der Compiler keinen sehr tiefen Einblick in Ihre Klassen und nimmt den leichten Weg: Er kopiert alle Daten 1:1.

Nehmen wir an, Sie haben folgende (nicht sehr nützliche) Klasse:

```
class Data {
    int *data;
public:
    Data(int size = 10) {
        data = new int[size];
```

```
    }

    ~Data() {
        delete [] data;
    }
};
```

Sieht soweit doch alles ganz gut aus. Aber wenn Sie Variablen dieses Typs einander zuweisen, wird es brenzlig:

```
Data a(2);
Data b;
// Was passiert hier?
b = a;
```

Da die Klasse keinen `operator=` hat, wird der standardmäßige Operator benutzt. Dieser kopiert alle Daten der Quelle in das Zielobjekt.

```
// Die Zuweisung
// b = a;
//entspricht so etwa
b.data = a.data;
```

Dadurch entstehen 2 Probleme. Zum einen kann der Speicher von `b.data` nie wieder freigegeben werden, da kein Zeiger mehr auf ihn vorhanden ist. Ein zweites Problem ist die Tatsache, dass nun `a.data` und `b.data` auf den gleichen Speicherbereich zeigen.

Wenn nun die Klassen ihren Gültigkeitsbereich verlassen, wird ihr Destruktor aufgerufen – und dadurch der gleiche Speicherbereich 2 mal freigegeben (einmal von `a`, einmal von `b`). Was zu einem Absturz Ihres Programms führen kann.

Und genau das ist so tückisch: Ein solcher Fehler kann Ihr Programm zum Absturz bringen. Wenn Sie Pech haben, stützt das Programm in diesem Moment noch nicht ab, da der Speicherbereich aus irgendeinem Grund geschickt liegt. Und in diesem Fall werden Sie den Fehler erst viel später bemerken und mit hoher Wahrscheinlichkeit Probleme haben, ihn auf das ursprüngliche Problem zurückzuführen.

Sie sollten also immer dafür sorgen, dass der Copy-Konstruktor und der `operator=` in jeder Klasse implementiert werden, die dynamischen Speicher benutzen.

```
class Data {
    int *data;
    int size;
```

```
public:
    Data(int size=10) {
        data = new int[size];
        this->size = size;
    }

    // Copy ctor
    Data(const Data& rhs) {
       copy(rhs);
    }

    Data& operator=(const Data& rhs) {
        if (this == &rhs) {
            return *this;
        }
        copy(rhs);
        return *this;
    }

    ~Data() {
        delete [] data;
    }

private:
    void copy(const Data&rhs) {
        if (data) {
            delete [] data;
        }
        size = rhs.size;
        data = new int[size];
        for (int a=0; a < size; a++) {
            data[a] = rhs.data[a];
        }
     }
};

int main(int, char**) {

    Data a;
    Data b(100);

    b = a;
    return 0;
}
```

Das eigentliche Kopieren der Daten wurde in die Methode `copy()` ausgelagert. Da diese Methode ein Implementierungsdetail ist, wurde sie `private` deklariert.

Stellen Sie sicher, dass alle Attribute kopiert werden

Auch dieser Tipp klingt im ersten Moment sehr trivial. Aber werfen Sie mal einen Blick auf folgende Klassen:

```
#include <iostream>

using namespace std;

class Base {
    int foo;

public:
    Base() : foo(-1) {
    }

    void setFoo(int value) {
        foo = value;
    }
    int getFoo() {
        return foo;
    }

    Base& operator=(const Base& rhs) {
        if (&rhs == this) {
            return *this;
        }
        foo = rhs.foo;
    }
};

class Child : public Base {
    int bar;
public:
    Child() : bar(-1){
    }

    void setBar(int value) {
        bar = value;
    }
```

```
    int getBar() {
        return bar;
    }
    Child &operator=(const Child& rhs) {
        if (this == &rhs) {
            return *this;
        }
        bar = rhs.bar;
    }
};

int main(int, char**) {
    Child a, b;
    a.setFoo(10);
    a.setBar(20);

    b = a;

    cout << "b.foo = " << b.getFoo() << "   b.bar=" << b.getBar() <<
endl;

    return 0;
}
```

Auf den ersten Blick sieht alles okay aus. Jedoch wird bei der Zuweisung von `Child foo` nicht gesetzt, wie Sie anhand der Ausgabe leicht erkennen können.

```
b.foo = -1   b.bar=20
```

Ein Zuweisungsoperator einer abgeleiteten Klasse muss auch die Datenelemente der Basisklasse zuweisen.

```
Child &operator=(const Child& rhs) {
    if (this == &rhs) {
        return *this;
    }
    ((Base&) *this) = rhs;
    bar = rhs.bar;
}
```

Diese etwas obskur wirkende Zeile konvertiert `*this` mittels eines cast in eine Referenz auf die Basisklasse und weist dann dieser den `rhs` Wert zu. Es ist wichtig, dass Sie `*this` auf eine Referenz der Basisklasse casten. Wenn Sie stattdessen nur in ein Objekt konvertieren, dann wird nicht direkt der `operator=` aufgerufen, sondern über den Copy-Konstruktor ein

neues Base Objekt erzeugt, und dann wird von diesem temporären Objekt der operator= aufgerufen. Nach dem Aufruf wird das temporäre Objekt wieder verworfen.

Wenn Ihre Basisklasse (wie in diesem Beispiel) einen eigenen Zuweisungsoperator definiert, dann können Sie es sich noch etwas einfacher machen:

```
Child &operator=(const Child& rhs) {
    if (this == &rhs) {
        return *this;
    }
    Base::operator=(rhs);
    bar = rhs.bar;
}
```

Die aktuelle Version des Gnu Compilers erlaubt es Ihnen auch (korrekterweise) den operator= aufzurufen, falls er in der Basisklasse nicht implementiert wurde, und stattdessen die vom Compiler erzeugte Variante zum Einsatz kommt.

Verbieten Sie vom Compiler erzeugte Methoden

Es gibt Fälle, in denen ein Copy-Konstruktor oder ein Zuweisungsoperator keinen Sinn machen. Wenn diese Methoden jedoch einfach nur weggelassen werden, implementiert der Compiler seine eigenen Varianten. Was nun?

Um zu verhindern, dass der Compiler eigene Methoden erzeugt müssen wir diese selbst implementieren. Und um sicher zu gehen, dass diese Methoden nicht aufgerufen werden können, deklarieren wir sie als private.

Ab diesem Zeitpunkt können nur noch Methoden der eigenen und friend Klassen auf den operator= zugreifen. Wenn wir dies auch noch verhindern wollen, dann definieren wir den Operator als private und implementieren ihn nicht. Auf diese Weise meldet der Linker einen Fehler, sobald der Zuweisungsoperator aufgerufen wird.

```
Child &operator=(const Child& rhs);
```

Konstruktoren globaler Variablen werden vor main() ausgeführt

Konstruktoren globaler Variablen werden vor main() ausgeführt. Dies ist normalerweise kein Problem, aber da wir in diesem Buch mit der Alle-

gro-Bibliothek arbeiten, und keine Allegro Funktion benutzt werden darf bevor diese Bibliothek initialisiert wurde ist es ein Punkt, der erwähnt werden muss.

```
#include <allegro.h>

class Buffer {
    BITMAP *data;
public:
    Buffer() {
        data = create_bitmap(SCREEN_W, SCREEN_H);
    }

    ~Buffer() {
        destroy_bitmap(data);
    }

    void show() {
        blit(data, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
    }
};

// Das gibt Ärger
Buffer doubleBuffer;

int main(int argc, char** argv) {
    allegro_init();

    // und so weiter
} END_OF_MAIN()
```

In einem solchen Fall sollten Sie einen Zeiger auf ein `Buffer` Objekt global definieren und dieses Objekt dann in `main()` zuweisen.

```
// So funktioniert es
Buffer *doubleBuffer;

int main(int argc, char** argv) {
    allegro_init();
    doubleBuffer = new Buffer();

    // und so weiter
} END_OF_MAIN()
```

9 Die Standard Template Library

Es gibt einige ständig wiederkehrende Datenstrukturen in Programmen. Damit nicht jeder Programmierer das Rad neu erfinden muss, wurde die ***Standard Template Library***, oder kurz: STL entwickelt. Sie enthält eine Sammlung äußerst nützlicher und vielseitig einsetzbarer Klassen.

Die Sammlung an Datencontainern reicht von Vektoren (Arrays mit dynamischer Größe) bis hin zu ausgewogenen binären Bäumen.

Mit anderen Worten: Wenn Sie Daten speichern müssen, dann brauchen Sie sich ab jetzt nur noch um die Daten zu kümmern, das Verwalten und Speichern übernimmt die STL für Sie.

Dieses Kapitel gibt eine Übersicht über die STL-Klassen und geht auf die am häufigsten benutzen Containerklassen ein. Nach dem Durcharbeiten des Kapitels werden Sie in der Lage sein, Container und Iteratoren so zu benutzen, als ob Sie noch nie etwas anderes getan hätten.

Schnelligkeit der STL

Die erste Frage, die gestellt wird, sobald es um Klassenbibliotheken geht, ist die nach der Geschwindigkeit. Wie schnell sind die Klassen im Zugriff? Ist mein eigener Code nicht schneller? Dies ist natürlich auch den Programmierern der STL bewusst und aus diesem Grund haben sie auch sehr viel Wert auf eine gute Performance gelegt. Auf Sicherheitsabfragen wurde weitestgehend verzichtet. So findet zum Beispiel keine Überprüfung statt, ob ein bestimmter Index (zum Beispiel bei einem Array Zugriff) gültig ist. Auch sind die benutzten Algorithmen ziemlich optimal umgesetzt worden und über die Jahre immer wieder verbessert worden. In den meisten Fällen wird also die STL nicht langsamer sein als von Ihnen geschriebener Code. Die Chancen stehen sogar nicht schlecht, dass die STL-Implementierung der jeweiligen Algorithmen schneller ist also Ihre eigene.

Grundlagen

Die Standard Template Library hat ihre eigenen Begriffe, und es ist wichtig, dass Sie mit diesen umgehen können. Aus diesem Grund gehe ich

hier auf die wichtigsten Punkte ein und erkläre, wie die einzelnen Elemente zusammenhängen.

Parameterübergabe

Innerhalb der STL werden alle Werte immer per Wert übergeben und niemals als Referenz. Bei primitiven Datentypen ist dies kein Problem, wenn Sie jedoch Objekte in den Containern speichern, dann sollten Sie sich überlegen, eventuell stattdessen Zeiger auf Objekte, oder Referenzen auf Objekte zu benutzen.

Container

Container bedeutet schlicht und ergreifend Behälter. Eine Container Klasse ist also eine Klasse, die als Behälter für andere Klassen dient und Methoden zur Verfügung stellt, um mit dem Inhalt zu arbeiten und ihn zu organisieren. Die STL stellt Ihnen folgende Container zur Verfügung:

- *Vector*: Ein dynamisches Array, das seine Größe anpassen kann, sobald der bestehende Platz nicht mehr reicht, um ein neues Element zu speichern.
- *Deque*: Eine Doppelschlange, oder auch Deck. Es handelt sich um eine lineare Datenstruktur, bei der sowohl am Anfang als auch am Ende Elemente hinzugefügt oder entfernt werden können. Ein Einfügen in der Mitte der Schlange ist jedoch nicht zulässig.
- *List*: Eine doppelt verkettete Liste. Sie können an jeder Position Elemente hinzufügen oder entfernen.
- *Stack*: Ein Stapel. Stellen Sie sich einen Kartenstapel vor. Sie können die oberste Karte vom Stapel nehmen oder eine Karte oben auf den Stapel legen. Andere Aktionen sind nicht möglich.
- *Queue*: Eine Schlange. Sie funktioniert so wie die Schlange an der Supermarktkasse. Am hinteren Ende stellen sich die Leute an, am vorderen Ende verlassen Sie die Schlange wieder.
- *Priority Queue*: Eine nach Wichtigkeit sortierte Schlange. Stellen Sie sich dies in etwa so vor wie beim Notarzt. Dringende Fälle werden weiter vorn in der Warteschlange plaziert, die nicht so dringenden Fälle weiter hinten. Bei gleich schweren Fällen entscheidet die Ankunftszeit.

- *Set*: Ein Set ist ein »Menge« im mathematischen Sinn. Sie können einer Menge Elemente hinzufügen, überprüfen ob ein Element in einer Menge enthalten ist oder sich alle Elemente der Menge nacheinander zurückgeben lassen.
- *Multiset*: Ein Multiset ist dem normalen Set sehr ähnlich. Der primäre Unterschied liegt darin, dass das gleiche Element in einem Multiset mehrfach vorkommen kann.
- *Map*: Eine Map ordnet einem bestimmten Element (dem Schlüssel, oder engl. »key«) ein anderes Element (den Wert, oder engl. »value«) eindeutig zu. Es ist also eine Art Verknüpfungstabelle. Sie können Werte mit einem Schlüssel verknüpfen, den zu einem Schlüssel gehörenden Wert erfragen oder sich alle Schlüssel zurückgeben lassen. Wichtig ist, dass die Zuordnung von Schlüssel zu Wert sehr schnell ist.
- *Multimap*: Bei einer Multimap können einem Schlüssel mehrere Werte zugeordnet sein, ansonsten verhält sie sich genauso wie eine normale Map.

Iteratoren

Iteratoren (iterators) erlauben es, die Elemente einer Collection nach einander zu bearbeiten. Die typischen Iteratoren, die in jeder Collection Klasse implementiert sind, sind `begin()` und `end()`, die das erste Element einer Collection und das Ende (also die Position nach dem letzten Element) bezeichnen.

Dieses Konzept ist sehr wichtig und zieht sich durch die gesamte STL. Sie können es sich so vorstellen, dass es in jedem Container immer noch ein weiteres Element gibt, das hinter den eigentlichen Daten steht und das Ende markiert. Auf diese Weise werden auch keine Sonderregeln mehr für leere Listen (oder andere Container) gebraucht. Wenn `begin() == end()`, dann ist der Container leer.

Bitte behalten Sie im Hinterkopf, dass Sie den `end()` Iterator niemals dereferenzieren dürfen. Das Ergebnis von `*end()` ist nicht definiert.

Wenn STL-Funktionen zwei Iteratoren übergeben werden müssen, dann wird der erste immer einschließlich, der zweite immer ausschließlich behandelt.

Sie können Iteratoren mit dem ++ Operator erhöhen (und damit zum nächsten Element wechseln) und mit dem Dereferenzierungsoperator * auf das Datenobjekt zugreifen, auf das der Iterator zeigt.

Algorithmen

Bei der STL wurde die Designentscheidung getroffen, Algorithmen nicht als Methoden der einzelnen Klassen zu implementieren. Stattdessen sind sie außerhalb definiert und arbeiten mit Iteratoren. Der Vorteil dieser Lösung ist, dass der gleiche Algorithmus auf quasi alle Container angewandt werden kann. Auch ist es so leicht möglich, neue Algorithmen zu schreiben, ohne bestehende Klassen überladen zu müssen.

Vektoren

Die Vektoren der Standard Template Library sind im Grunde nichts anderes als in der Größe veränderbare Arrays. Obwohl es nicht genau festgelegt ist, in welcher Art und Weise die Daten tatsächlich gespeichert werden, können Sie davon ausgehen, dass ein Vektor in den meisten Fällen intern ein Array zur Datenhaltung benutzt.

Auch vom Verhalten her sind Vektoren den normalen Arrays sehr ähnlich. Sie können sogar einen Vektor behandeln wir ein Array:

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> IntVector;
typedef IntVector::iterator IntVectorIterator;


int main (int , char *) {

    IntVector iv;
    iv.reserve(20);
    iv.resize(2);
    iv[0] = 10;
    iv[1] = 20;
```

```
    cout << iv[0] << "  " << iv[1] << endl;

    return(0);
}
```

Am Anfang definieren wir zwei neue Typen für den Vector und den darauf arbeitenden Iterator. Dies ist zwar nicht zwingend notwendig, erhöht aber die Lesbarkeit des Quelltextes deutlich.

Dann wird ein `IntVector` erzeugt und durch `reserve()` Platz für 20 Elemente angelegt. Als Nächstes teilen wir dem Vector mit, dass wir die ersten 2 Einträge benutzen wollen. Dann weisen wir den ersten beiden Elementen Werte zu und geben diese aus.

Es ist wichtig, den Unterschied zwischen `reserve()` und `resize()` zu verstehen. Ein Vektor wird in den meisten Fällen etwas mehr Speicher anlegen als er in diesem Moment wirklich braucht. Der Grund dafür ist, dass das Anlegen von Speicher und das Kopieren der Daten in den neuen Speicherbereich recht lange dauert. Also legt der Vektor mehr Speicher an, und kann dann auf diesen Bereich zugreifen, sobald er gebraucht wird.

Die Anzahl der tatsächlich gespeicherten Werte ist die Größe des Vektors. Sie wird durch `size()` ermittelt und durch `resize()` geändert. Die Anzahl der Werte, die im Array gespeichert werden können bevor neuer Speicher angelegt werden muss, wird als Kapazität bezeichnet. Sie können die Kapazität durch `reserve()` verändern und durch `capacity()` abfragen.

```
#include <vector>
#include <iostream>

using namespace std;

int main(int , char**) {

    vector<int> intVec;

    for (int a=0; a < 9; a++) {
        intVec.push_back(a);
        cout << "size: "     << intVec.size()  << "   ";
        cout << "capacity: " << intVec.capacity()  << endl;
    }

    return 0;
}
```

Mit diesem kleinem Testprogramm können Sie testen, wie sich Größe und Kapazität ändern, wenn Werte zum Vektor hinzugefügt werden. Ein häufig benutztes Schema ist es, den Speicher zu verdoppeln, sobald der bestehende nicht mehr ausreicht. Bei der STL-Version meines Compilers sieht die Ausgabe des oben abgebildeten Programms wie folgt aus:

```
size: 1   capacity: 1
size: 2   capacity: 2
size: 3   capacity: 4
size: 4   capacity: 4
size: 5   capacity: 8
size: 6   capacity: 8
size: 7   capacity: 8
size: 8   capacity: 8
size: 9   capacity: 16
```

Sie sehen, sobald die `size` die `capacity` erreicht, verdoppelt sich diese. Von 1 auf 2, dann 4, 8 und schließlich 16. Sie sollten sich immer bewusst sein, ob ein Hinzufügen von Elementen den Vektor vergrößert, da dies alle bestehende Iteratoren ungültig macht.

Bildliches Beispiel für Vektoren

Stellen Sie sich den Vektor als eine Art Setzkasten vor, bei dem jedes Fach nummeriert ist. Wenn Sie etwas aus dem Fach 5 haben wollen, dann können Sie einfach in dieses Fach greifen und den Gegenstand herausnehmen. Ebenso einfach können Sie in ein beliebiges Fach einen Gegenstand stellen.

Wenn Sie jedoch die Reihenfolge der Elemente ändern, zum Beispiel weil Sie einen bestimmten Gegenstand im ersten Fach des Setzkastens haben wollen, dann müssen Sie alle anderen Gegenstände versetzen.

Sollte der Platz nicht ausreichen, dann müssen Sie neue Fächer anbauen. Stellen Sie sich das so vor, dass Sie sich in diesem Fall mit etwas Holz, Leim und einer Laubsäge in Ihren Bastelkeller setzen und einen größeren Setzkasten bauen. Dann sortieren Sie alle Gegenstände vom alten in den neuen Kasten um, und verbrennen schließlich den alten.

Der in der Abbildung 9.1 gezeigte Vektor hat eine Kapazität von 30 Elementen, und eine Größe von 4.

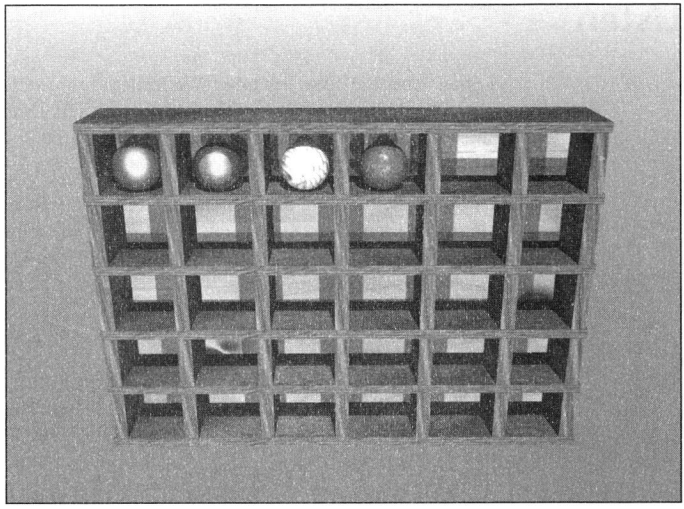


Abbildung 9.1: Ein Vektor bildlich dargestellt

Einsatz von Vektoren

Der große Vorteil von Vektoren ist, dass Sie jedes Element direkt ansprechen können. Brauchen Sie also diesen direkten Zugriff, dann sind Vektoren der Datentyp Ihrer Wahl. Behalten Sie aber auch im Hinterkopf, dass beim Einfügen eines Elements in der Mitte des Vektors alle dahinter liegenden Elemente verschoben werden müssen. Fügen Sie ein Element am Beginn des Vektors ein, dann müssen alle Elemente verschoben werden.

Der Idealfall für die Benutzung eines Vektors ist, wenn Sie in etwa wissen wie viele Elemente Sie brauchen, es aber in Ausnahmefällen auch mal mehr werden können. Kombinieren Sie dies mit wahlfreiem Zugriff und dass neue Elemente meist am Ende angehängt werden, und schon haben Sie die ideale Situation, einen Vektor einzusetzen.

Oder in anderen Worten: Wenn Sie ein Array mit ein paar Komfortfunktionen brauchen, dann ist der Vektor die erste Wahl.

Listen

Listen sind die Eier legenden Wollmilchsäue unter den Containerklassen. Wenn Sie sich nicht sicher sind, welchen Datentyp Sie brauchen, sind die Chancen recht hoch, dass Sie mit einer Liste recht gut bedient sind.

Die STL-Listen sind als doppelt verkettete Listen realisiert. Das bedeutet: Jedes Element kennt seinen Vorgänger und seinen Nachfolger, kennt aber nicht seine absolute Position innerhalb des Containers. Wenn Sie also den 15ten Eintrag wollen, dann müssen Sie beim ersten Element beginnen, von da zum nächsten Element gehen, und wieder zum nächsten, und so weiter, bis Sie dann schließlich beim 15ten Eintrag angekommen sind.

Diesen Nachteil machen Listen dadurch wieder wett, dass Sie an jeder beliebigen Position sehr schnell Elemente einfügen und entfernen können.

```
#include <list>
#include <iostream>

using namespace std;

typedef list<int> IntList;
typedef IntList::iterator IntListIterator;
typedef IntList::reverse_iterator
                                IntListReverseIterator;

int main(int , char**) {

    IntList intList;

    // Am Ende was anhängen
    for (int a=10; a < 19; ++a) {
        intList.push_back(a);
    }

    // Element am Anhang hinzufügen
    for (int a=1; a < 10; ++a) {
        intList.push_front(0-a);
    }
```

```
    // Vorwärts durch die Liste
    for (IntListIterator cur = intList.begin();
                         cur != intList.end(); ++cur) {
        cout << *cur << endl;
    }
    cout << "------" << endl;
    // Rückwärts durch die Liste
    for (IntListReverseIterator cur = intList.rbegin();
                        cur != intList.rend(); ++cur) {
        cout << *cur << endl;
    }

    // Alle geraden Werte löschen
    // Beachten Sie, dass am Ende des for() kein
    // ++cur steht.
    for (IntListIterator cur = intList.begin(); cur !=
                                    intList.end(); ) {
        if (*cur % 2 == 0) {
            // Wenn Sie Zeiger auf Objekte speichern,
            // dann sollten Sie hier das Objekt deleten
            cur = intList.erase(cur);
        }  else {
            // ungerade, also einfach weiter
            ++cur;
        }
    }

    // Und nochmal ausgeben
    cout << endl << "Nur die ungeraden Werte:" << endl;
    for (IntListIterator cur = intList.begin(); cur !=
                                intList.end(); ++cur) {
        cout << *cur << endl;
    }

    return 0;
}
```

Am Anfang des Testprogramms definieren wir einige Datentypen für die Liste und die auf der Liste arbeitenden Iteratoren. Der Grund hierfür ist in erster Linie Lesbarkeit. Die Template Schreibweise ist auf Dauer doch etwas unübersichtlich. Falls Sie `IntListReverseIterator cur;` für unübersichtlich halten, dann bedenken Sie, dass die Alternative `list< int >::reverse_iterator cur;` ist (und dass Sie ja durchaus auch einen kürzeren Namen wählen können).

Nachdem die Liste erstellt wurde, fangen wir direkt an, Daten einzufügen. Es ist nicht notwendig vorher Speicher zu reservieren, sondern wir können direkt Daten anhängen.

Die erste Schleife hängt 10 Werte an das Ende der Liste, die zweite hängt 10 Werte vor den Anfang der Liste. Beide Schleifen sollten gleich schnell ablaufen, da keinerlei Daten verschoben werden müssen.

Da keine Daten verschoben werden, bleiben auch die bereits bestehenden Iteratoren gültig.

Nun wird die Liste einmal vorwärts ausgeben. Dazu benutzen wir den am Anfang definierten Iterator. Mit `begin()` bekommen wir einen Iterator, der auf das erste Element der Liste zeigt, `end()` liefert einen Iterator zurück, der hinter das letzte Element zeigt. Wir kommen zur nächsten Position, in dem wir den Iterator erhöhen. Um auf das Element zuzugreifen, auf das ein Iterator zeigt, nutzen Sie den Dereferenzierungsoperator *. Iteratoren verhalten sich also in etwa so wie Zeiger auf den Datentyp, den der Container speichert.

Wollen Sie rückwärts durch die Liste gehen, dann steht Ihnen zu diesem Zweck der `reverse_iterator` zu Verfügung. Dieser erlaubt es Ihnen auf genau die gleiche Weise, rückwärts wie vorwärts durch die Liste zu gehen. `rbegin()` gibt einen Iterator auf das letzte Element der Liste zurück, `rend()` einen Iterator, der vor dem ersten Element der Liste steht.

Zum Abschluss noch ein recht häufig vorkommendes Beispiel: Das Überprüfen der Liste auf Elemente, die entfernt werden müssen (Stellen Sie sich vor die Liste enthält die feindlichen Raumschiffe in einem Ballerspiel. Sobald der Spieler ein Raumschiff abschießt muss es aus der Liste und damit auch vom Schirm entfernt werden.). Die `erase()` Methode entfernt ein Element und gibt den nächsten Iterator in der Folge zurück. Bei einem normalen, vorwärtsschreitenden Iterator also das nächste Element, bei einem rückwärtsschreitenden Iterator das vorhergehende Element.

Natürlich können Sie nicht nur Elemente entfernen, sondern ebenso Elemente an einer bestimmten, durch einen Iterator spezifizierten Stelle einfügen.

Bildliches Beispiel für eine Liste

Stellen Sie sich vor, Sie knoten zwei Schnürsenkel an einem Ende zusammen und halten die beiden verbleibenden Enden in jeweils einer Hand. Dies entspricht der leeren Liste. Die beiden Schnürsenkel entsprechen dem Iterator für das Element vor dem ersten Element in der Liste (`rend()`) und dem Iterator für das Element nach dem letzten Eintrag in der Liste (`end()`).

Wenn Sie nun ein Element hinzufügen, dann binden Sie dieses erst an einem neuen Schnürsenkel fest, öffnen dann den Knoten, der die beiden bisherigen Senkel verbindet, und binden den neuen Schnürsenkel dazwischen.

Wenn Sie zu einem bestimmten Element in der Liste wollen, dann stellen Sie sich vor, dass Sie diese Kette von Schnürsenkeln in einem dunklen Raum halten. Um nun ein bestimmten Eintrag zu finden, müssen Sie die Kette Knoten für Knoten abtasten und sich die aktuelle Nummer merken.

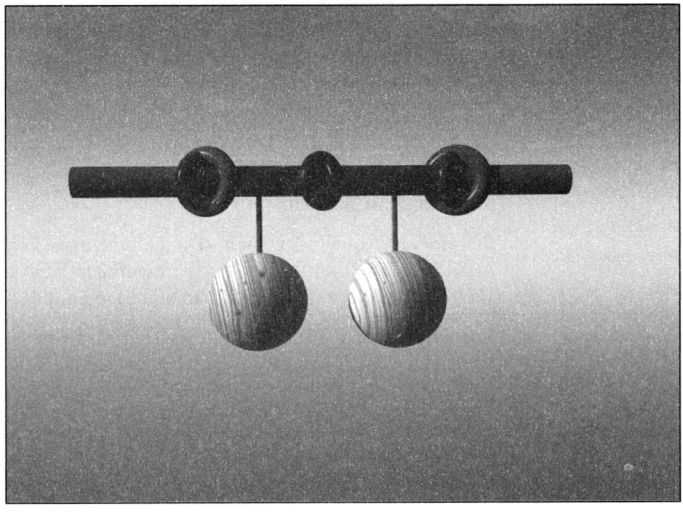


Abbildung 9.2: Eine Liste mit 2 Elementen

Einsatz von Listen

Wie bereits anfangs erwähnt, sind Listen mit die flexibelste Datenstruktur. Sie können Daten an jeder beliebigen Stelle sehr schnell einfügen und/oder entfernen. Jedoch können Sie diese Stelle nicht direkt anspringen.

Auch brauchen Sie keinen Speicher zu reservieren. Nur für die Menge an Elementen, die auch wirklich gebraucht wird, wird Speicher reserviert. Dies bedeutet aber nicht, dass eine Liste immer weniger Speicher benötigt als ein Array oder Vektor. Intern werden für jedes Element zwei Zeiger gespeichert (um auf das vorhergehende bzw. nächste Element zugreifen zu können). In dem oben skizzierten Fall einer Liste von `int` Elementen braucht also jeder Eintrag doppelt so viel Speicher wie das zu speichernde Element selbst.

Deques und Queues

Eine Queue, oder auch Schlange stellen Sie sich am besten wie die Warteschlange im Kino vor: Wer zuerst kommt, wird auch zuerst bedient, wer neu dazukommt, muss sich am Ende der Schlange anstellen, und warten, bis er an der Reihe ist.

Mit anderen Worten: Sie können Daten am Ende der Schlange hinzufügen (`push_back()`) und Daten vom Anfang der Schlange entfernen (`pop_front()`), ohne bei diesen Aktionen eine Geschwindigkeitseinbuße hinnehmen zu müssen.

Bei einem Deque, oder auch »Doppelschlange« können Sie sowohl am Anfang als auch am Ende Daten hinzufügen bzw. entfernen. Sie haben auch jederzeit direkten Zugriff auf alle Element des Deques.

Aus funktionaler Sicht handelt es sich also um eine Erweiterung der Schlange, aus Effizienzgründen wird das Deque jedoch meist anders implementiert, um einen schnellen direkten Zugriff zu gewährleisten.

```
#include <deque>
#include <queue>
#include <iostream>

using namespace std;

int main(int , char**) {
```

```
    deque<int> id;

    cout << "Deque Test" << endl;
    id.push_back(10);
    id.push_back(20);
    id.push_front(5);
    id.push_front(1);

    for (deque<int>::iterator i = id.begin();
                                i != id.end(); ++i) {
        cout << *i << endl;
    }
    cout << "------------" << endl;
    for (int a = 0; a < id.size(); a++) {
        cout << id[a] << endl;
    }
    cout << "------------" << endl;

    while (id.size() > 0) {
        cout << id.front() << endl;
        id.pop_front();
        cout << id.back() << endl;
        id.pop_back();
    }

    queue<int> iq;
    cout << "Queue Test" << endl;

    iq.push(100);
    iq.push(200);

    cout << iq.front() << endl;
    iq.pop();
    cout << iq.front() << endl;
    iq.pop();

    return 0;
}
```

In diesem Beispiel habe ich bewusst auf die Deklaration von neuen Typen für die Deque- und Queue-Variante für `int` verzichtet, damit Sie sich selbst ein Bild davon machen können, welche Variante Ihnen lieber ist.

Das Beispiel beginnt mit der Deklaration eines `int` Deques namens `id`. An dieses Deque werden dann 2 Zahlen angehängt und dann 2 Zahlen

vorangestellt. Die Ausgabe des Inhalts erfolgt auf die altbekannte Weise: Eine `for` Schleife, in der mittels eines Iterators jedes Element ausgegeben wird. Um zu zeigen, dass auch ein direkter Zugriff auf die einzelnen Elemente möglich ist, wird dann der komplette Inhalt nochmals ausgegeben, diesmal jedoch mit einer »gewöhnlichen« `for` Schleife.

Schließlich werden die Elemente wieder entfernt. Interessant hierbei ist, dass die `pop_front()` und `pop_back()` Methoden nicht das Element zurückliefern, welches Sie gerade entfernt haben, sondern das Element einfach nur entfernen. Die STL trennt hier klar das Entfernen des Elements von der Abfrage des ersten bzw. letzten Elements.

Bei den Pop-Operationen behalten alle Iteratoren, die nicht auf das entfernte Element zeigen ihre Gültigkeit.

Bei der Queue haben Sie nur eine Möglichkeit Daten anzuhängen. Aus diesem Grund gibt es auch keine `push_back()` Methode. Um Daten anzuhängen, benutzen Sie einfach nur `push()` um Daten zu entfernen, reicht ein `pop()`.

Maps

Stellen Sie sich vor, Sie können Ihren Daten einen Namen geben. Anstatt über die Position innerhalb einer Datenstruktur auf das Objekt zuzugreifen, nehmen Sie einfach den entsprechenden Namen.

```
// Zugriff über Position
flags[PRINCESS_SAVED] = true;
// Zugriff mittels Namen
flags["Princess saved"] = true;
```

Und genau das ist durch den Einsatz von Maps möglich.

Der Unterschied mag im ersten Moment nicht so gravierend erscheinen, kann aber Ihre Arbeit deutlich erleichtern. Stellen Sie sich vor, Sie wollen eine Debug Konsole in Ihr Spiel einbauen, um beim Testen bestimmte Variablen ändern zu können. Dieses Kommando können Sie mit Hilfe von Maps sehr einfach implementieren, da Sie den eingegebenen Text direkt als Schlüssel verwenden können.

```
#include <map>
#include <string>
#include <iostream>
```

```
using namespace std;

int main(int , char**) {

    map<string, string> vars;
    map<string, string>::iterator pos;

    string input;

    do {
        // Erstes Wort nach input
        cin >> input;
        if (input == "set") {
            // Befehl ist "set"
            string var;
            string value;

            // Syntax ist
            // var = value
            cin >> var;
            cin >> input;
            cin >> value;

            if (input != "=") {
                cout << "Syntax:" << endl
                     <<"set var = value" << endl;
            } else {
                // speichern des wertes
                vars[var] = value;
                cout << "*ok" << endl;
            }
        } else if (input == "echo") {
            // Echo Kommando

            // Variablen Namen einlesen
            cin >> input;
            // Und in der Map suchen
            pos = vars.find(input);

            if (pos != vars.end()) {
                // Gefunden -> also ausgeben
                cout << pos->second << endl;
            } else {
                // Nicht gefunden
                cout << "*variable " << input
```

```
                    <<" not found." << endl;
            }
        }
    } while (input != "exit");

    return 0;
}
```

Dieses kurze Programm erlaubt es Ihnen, Variablen mit beliebigem Namen und Inhalt anzulegen und diese dann wieder ausgeben zu lassen.

Aber der Reihe nach: Nach den üblichen `includes` wird die Map und ein Iterator für die Map deklariert. In der Schleife wird dann eine Eingabe in der Variablen `input` gespeichert. Dann wird diese Eingabe mit den vorhandenen Befehlen (»set« und »echo«) verglichen.

Wurde »set« eingegeben, dann werden 3 weitere Wörter von `cin` gelesen: der Variablenname, der Zuweisungsoperator und der eigentlich Wert. Zwar ist dies nicht sonderlich flexibel (da zum Beispiel immer ein Leerzeichen die einzelnen Wörter trennen muss), reicht aber zu Demonstrationszwecken.

Entspricht die Eingabe den Syntaxanforderungen, dann wird ein neuer Eintrag in die Map hinzugefügt und eine Bestätigung für den Nutzer ausgegeben. Im Falle eines Syntaxfehlers wird die korrekte Syntax für den »set« Befehl angezeigt.

Mit diesen wenigen Zeilen können wir schon Variablen erstellen und speichern. Mit dem zweiten Befehl »echo« können wir die gespeicherten Daten dann auch wieder abrufen.

Zuerst wird der Name der auszugebenden Variable eingelesen und nach einem passenden Eintrag in der Map gesucht. Die `find()` Methode der Map liefert einen Iterator zurück, der auf den gefunden Eintrag zeigt. Wird der angegebene Schlüssel nicht gefunden, dann liefert `find()` den `end()` Iterator zurück.

Da jeder Eintrag der Map aus 2 Elementen besteht, können wir nun nicht mehr einfach mit dem * Operator auf das Datenelement zugreifen. Statt dessen müssen wir die beiden Variablen `first` und `second` benutzen. Die Variable `first` zeigt auf den Schlüssel, die Variable `second` auf den zugeordneten Eintrag.

Wird der entsprechende Eintrag gefunden (also wenn `pos` ungleich `end()` ist), dann gibt das Programm den gefundenen Wert aus. Ansonsten wird

eine Fehlermeldung ausgegeben, damit der Benutzer weiß, dass die angegebene Variable bisher noch nicht existiert.

Speichern Sie Zeiger

Es gibt einen Punkt, auf den Sie unbedingt achten sollten: Die STL übergibt alle Werte »by Value« – das heißt die Daten werden kopiert. Bei einfachen Datentypen ist dies kein Problem, aber wenn Sie komplexere Strukturen oder Klassen speichern wollen, dann sollten Sie Zeiger vorziehen.

```
struct Sprite{
    float  x,  y;
    float dx, dy;

    int   frameCount;
    int   animationCount;
    int   **animation;
    int   curAnimation;
    int   curFrame;

    BITMAP *frames;
};
```

Diese Struktur hat eine Größe von 40 Bytes. Wenn Sie nun diese Struktur in einer Liste speichern wollen, dann würden jedes Mal, wenn Sie ein Sprite zu der Liste hinzufügen, 40 Bytes kopiert werden. 40 Bytes klingt nach nicht sehr viel, es kann Ihr Spiel aber deutlich ausbremsen wenn Sie viele dieser Objekte haben, und sie häufig erzeugt, zur Liste hinzugefügt und dann wieder entfernt werden.

Bei größeren Strukturen (die zum Beispiel einen Puffer für Daten, Farbwerte oder Zeichenketten enthalten könnten) ist dies noch extremer. Die Lösung für dieses Problem ist recht einfach (und wurde auch schon im Titel verraten): Speichern Sie Zeiger auf Ihre Daten, anstatt die Daten in die Liste zu kopieren.

```
typedef list<*Sprite> SpriteList;
int main(int, char**) {
    Sprite *spr;
    SpriteList list;
    spr = new Sprite();
    list.push_back(spr);
}
```

Functors

Operationen, die mit den Collection-Klassen arbeiten, nehmen meist einen so genannten Functor als Parameter. Ein Functor ist eine Klasse, bei der der `operator ()` überladen wurde. Dies erlaubt es, die Klasse wie eine Funktion aufzurufen.

```
#include <iostream>

class Test {
    public:
        void operator()() {
            std::cout << "Test";
        };
};

int main(int, char**) {
    Test t;
    t();

}
```

Lassen Sie sich nicht von den beiden aufeinander folgenden Klammerpaaren verwirren. Beim Überladen von Operatoren folgt auf das Schlüsselwort `operator` immer der Operator, der überladen wird, dann eine Klammer mit den Parametern, die der Operator bekommt. In diesem speziellen Fall ist aber der zu überladende Operator die Klammer, was zu der irritierenden Doppelklammer führt. Wenn wir das obige Beispiel um eine überladenen `operator()` erweitern, der ein `int` als Parameter übergeben bekommt, wird dies vielleicht deutlicher.

```
#include <iostream>


using namespace std;

class Test {
    public:
        void operator()() {
            cout << "Test" << endl;
        }
        void operator()(int a) {
            for (int i=0; i < a; i++) {
                cout << "Test" << endl;
```

```
            }
        }
};

int main(int, char**) {
    Test t;
    t();
    cout << "------------" << endl;
    t(3);
}
```

Die STL benutzt Functors als Möglichkeit für den Programmierer, auf Algorithmen Einfluss zu nehmen. Nehmen wir an, wir haben eine Liste von Rollenspielcharakteren. Mit Hilfe von Functors und der `sort()` Methode der `std::list` können wir diese beliebig sortieren.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

struct RPGChar {
    string name;
    int    level;
    int    attack;
    int    defense;
    int    magic;

    RPGChar(const char *heroName,
            const int   lvl, const int   atk,
            const int   def, const int   mgc)
        // initialisierungs liste
        : name(heroName),
          level(lvl), attack(atk),
          defense(def), magic(mgc)  {
    }
};

class SortByLevel {
    public:
    bool operator() (RPGChar* lhs, RPGChar* rhs) {
        // Sortieren nach Level, attack, defense
        // und magic (in dieser Reihenfolge)
        if (lhs->level < rhs->level) {
```

```
            return true;
        }
        if (lhs->attack < rhs->attack) {
            return true;
        }
        if (lhs->defense < rhs->defense) {
            return true;
        }
        if (lhs->magic < rhs->magic) {
            return true;
        }
        return false;
    }
};

class SortByAttack {
    public:
    bool operator() (RPGChar* lhs, RPGChar* rhs) {
        if (lhs->attack < rhs->attack) {
            return true;
        }
        return false;
    }
};
void class Remover {
    public:
    bool operator() (RPGChar *value) {
        delete value;
        return true;
    }
};

class Lister {
    public:
        bool  operator()(RPGChar* rpgChar) {
            cout
            << "Name:    " << rpgChar->name    << endl
            << "Level:   " << rpgChar->level   << endl
            << "Attack:  " << rpgChar->attack  << endl
            << "Defense: " << rpgChar->defense << endl
            << "Magic:   " << rpgChar->magic   << endl
            << endl;
            return true;
        }
};
```

```
int main(int, char**) {
    list<RPGChar*> rpgList;

    RPGChar *colt   = new RPGChar("Colt"  ,5,10,7,1);
    RPGChar *cherry = new RPGChar("Cherry",6,5,6,5);
    RPGChar *gustav = new RPGChar("Gustav",8,9,9,2);
    RPGChar *mara   = new RPGChar("Mara"  ,7,5,8,1);

    rpgList.push_back(colt);
    rpgList.push_back(cherry);
    rpgList.push_back(gustav);
    rpgList.push_back(mara);

    rpgList.sort(SortByLevel());
    for_each(rpgList.begin(), rpgList.end(), Lister());

}
```

Am Anfang des Listings steht die `RPGChar`-Struktur, die nichts anderes tut als die Daten für dieses Beispiel zur Verfügung zu stellen. Dann folgen zwei Functor-Klassen, die zur Sortierung dienen. Diese Arten von Funktoren nehmen zwei Parameter und geben ein `bool` zurück. Für die normale, aufsteigende Sortierung muss `true` zurückgegeben werden, wenn der erste Parameter kleiner ist als der zweite. Achten Sie darauf, dass der Typ der übergebenen Parametern auch der dem Container verwendeten Typ entspricht.

Der `Lister` Functor gibt ein einzelnes RPGChar-Objekt aus.

Der `Remover` Functor gibt ein Objekt wieder frei.

Unter Berücksichtigung des letzten Tipps werden in diesem Beispiel auch Zeiger auf `RPGChars` benutzt.

Das eigentliche Hauptprogramm ist recht simpel. Es werden erst ein paar Helden angelegt und dann zur Liste hinzugefügt.

Ein Aufruf von `sort()` mit einer der beiden Sortierungsfunktoren `SortByLevel` und `SortByAttack` sortiert die Liste in der durch diese Funktoren festgelegten Weise.

Für die Ausgabe der Liste verwenden wir einen von der STL zur Verfügung gestellten Funktor: `for_each()`. Sie übergeben diesem Funktor den Start und Enditerator und einen Funktor, der für jedes Element in diesem Bereich aufgerufen wird.

Die STL definiert Namen für bestimmte Typen von `Functors`.

Name	Beschreibung
Generator	Keine Parameter, gibt einen selbstgewählten Typ zurück. Wird zum Beispiel von generate() benutzt, um einen Container mit Werten zu füllen.
Unary Function	Hat einen Parameter und einen selbstgewählten Rückgabetyp. Wird zum Beispiel von for_each() benutzt, um eine Funktion für jedes Element aufzurufen.
Predicate	Der Sonderfall einer Unary Function, bei der der Rückgabewert vom Typ bool ist. Wird zum Beispiel von find_if() benutzt, um ein Element zu finden, bei dem eine bestimmte Bedingung erfüllt ist.
Binary Function	Hat zwei Parameter, und einen selbstgewählten Rückgabetyp. Wird zum Beispiel von accumulate() benutzt, um die in einem Container enthaltenen Elemente aufzuaddieren.
Binary Predicate	Der Sonderfall einer Binary Function, bei der der Rückgabewert vom Typ bool ist. Wird zum Beispiel von search() verwendet.

Tabelle 9.1: Functor-Typen

Diese Terminologie eignet sich hervorragend, um eine einfache Sache komplizierter klingen zu lassen. Stören Sie sich nicht daran. Merken Sie sich einfach nur, dass ein Predicate immer ein `bool` zurückliefert. Die Anzahl der Parameter ist meist offensichtlich und stellt aus diesem Grund kein Problem dar.

Abschließende Bemerkungen

Die STL kann Ihnen das Leben deutlich einfacher machen. Wenn Sie zuvor noch nicht mit ihr gearbeitet haben, ist jedoch eine gewisse Einarbeitungszeit notwendig. Natürlich ist dieses Kapitel keine umfassende Einführung in die Standard Template Library – damit könnte man ein eige-

nes Buch füllen. Es sollte Ihnen aber genug Basiswissen geben, um die wichtigsten Klassen in eigenen Projekten verwenden zu können. Wird in den späteren Kapiteln ein bisher noch nicht angesprochenes Feature der STL benutzt, wird dieses dort im Kontext erklärt.

Aber am schnellsten lernen Sie den Umgang mit den Template-Klassen, wenn Sie sich hinsetzen und ein paar Dinge ausprobieren. Werfen Sie einen Blick auf die Klassenübersicht der STL und experimentieren Sie herum. Schreiben Sie eigene `Functor` Klassen, damit dieses Konzept für Sie selbstverständlich wird. Die STL bietet Ihnen viele Möglichkeiten und ist aufgrund der schieren Menge an Klassen doch etwas unübersichtlich. Lassen Sie sich davon nicht abschrecken. Wie immer gilt auch hier: Übung macht den Meister.

10 Allegro

Allegro ist eine Bibliothek für die Spiele Programmierung – und was für eine! Allegro unterstützt den Programmierer mit einer Vielzahl von Routinen, die das Spiele-Entwicklerleben einfacher machen. Vom Setzen des Grafikmodus über die Anzeige und Manipulation von Grafiken bis hin zum Verwalten von Spieldaten: All dies ist in Allegro enthalten. Und für die Dinge, die nicht bereits direkt unterstützt werden, gibt es eine Vielzahl von Erweiterungen.

Dieses Kapitel gibt einen kurzen Überblick über Allegro. Welche Vorteile es hat, welche Nachteile damit verbunden sind, welche Betriebssysteme unterstützt werden und vieles mehr.

Allegro finden Sie auf der Buch-CD.

Was ist Allegro?

Das Wort »Allegro« bedeutet in der Musik »schnell, lebendig und hell«. Es ist außerdem eine rekursive Abkürzung für ***Allegro Low Level Game ROutines***.

Allegro entstand auf dem Atari ST, aber leider ging die Goldene Ära der Heimcomputer viel zu schnell zu Ende. Nach dem Atari ST war der PC die Plattform der Wahl für Allegro. Der Gnu C Compiler für DOS (djgpp) war die primäre Entwicklungsplattform während dieser Zeit. Recht schnell wurde Allegro dann von Nutzern des GCC auf anderen Plattformen übernommen, die den Source Code für Plattformen wie das X-Windows System oder Microsofts DirectX anpassten und eigene Allegro Versionen erstellten.

Inzwischen sind all diese (und noch weitere) Plattformen unter einen Hut zusammengefasst.

Unter Windows unterstützt Allegro neben DirectX auch das GDI, das ***Graphics Device Interface***. Das GDI ist die Schnittstelle, die »normale« Applikationen benutzen, um Grafiken auf dem Bildschirm darzustellen. Dies erlaubt es einem, Allegro nicht nur für Spiele, sondern auch für normale Applikationen einzusetzen.

Plattformübergreifende Spiele

Spiele, die mittels Allegro entwickelt wurden, können ohne Änderung auf einer Vielzahl von Plattformen kompiliert werden. Dazu gehören: Windows, Linux, MacOS, BeOS, DOS, OS/2, SunOS/ Solaris, FreeBSD, QNX und noch ein paar mehr.

Unter Windows werden alle populären Compiler unterstützt: MS Visual C, Borland C++ und MinGW (Minimalist Gnu for Windows).

In diesem Buch wird der Schwerpunkt auf dem Gnu C Compiler liegen. Dieser ist für jede der oben genannten Plattformen erhältlich, frei verfügbar und kostenlos. Die Windows Version ist darüber hinaus auf der CD enthalten, Linux und BSD User sollten bereits eine Version installiert haben.

In den Beispielen im Buch wird größtenteils auf die Windows Version eingegangen, die Unterschiede zwischen den Plattformen sind aber sehr gering. So gibt es in den Makefiles der Windows und Linux Versionen nur 2 Unterschiede: Ausführbare Dateien haben unter Windows eine ».exe«-Erweiterung und die Methode zur Einbindung der Bibliothek ist minimal anders.

Funktionalität

Neben der Unterstützung für die oben genannte Plattformvielfalt, hat Allegro noch andere Features.

Grafik

- ✔ Grundkörper wie Rechtecke, Kreise, Ellipsen und Vielecke (Polygone).
- ✔ Bitmap-Grafiken und Sprites, die gedreht, gestaucht, gespiegelt und mit Transparenzeffekten dargestellt werden können. Neben normalen Sprites werden auch RLE Sprites und kompilierte Sprites unterstützt.

- ✔ Textausgabe für Fonts mit fester oder proportionaler Größe. Unterstützung für mehrfarbige Fonts.
- ✔ Setzen aller von der Hardware unterstützter Vollbild und Fenster Modi

Musik und Sound

- ✔ Soundausgabe auf bis zu 64 Kanälen wird unterstützt. Alle Geräusche können in einer Schleife oder auch nur einmalig abgespielt werden.
- ✔ Soundeffekte können rückwärts gespielt werden
- ✔ Die Geschwindigkeit, Lautstärke und Position im Raum können noch während des Abspielens geändert werden.
- ✔ MIDI Songs können normal oder in einer Schleife abgespielt werden.
- ✔ Aufnahme-Funktionalität für Sound und MIDI.
- ✔ Support für Software Wavetable MIDI Playback.

Eingabegeräte

- ✔ Alle vom System unterstützten Joysticks können mühelos in Allegro angesprochen werden.
- ✔ Mausunterstützung. Neben normaler Rückgabe der Mauszeigerposition können auch die Hardware-nahen Informationen der Maus abgefragt werden (Positionsänderung in Mickeys). Unterstützung für das Mausrad.
- ✔ Die Tastatur kann entweder konventionell oder als »Joypad mit 112 Tasten« abgefragt werden.

Sonstiges

- ✔ Timerfunktionen können auf einfache und Plattform-übergreifende Weise erstellt werden
- ✔ Unterstützung für LZSS-Komprimierung/-Dekomprimierung
- ✔ Unterstützung für Multi-Objekt Resourcefiles.

- Schnelle mathematische Funktionen, inklusive einer Look-Up-Table-gestützten Variante von Sinus und Cosinus und Fix-Punkt-Dezimal Operationen.
- Unterstützung für Konfigurations- (INI) Dateien.

Neben dieser Menge an unterstützten Funktionen hat Allegro auch noch einige weitere Vorteile. Zum einen ist der Quell Code offen. Dies erlaubt es einem, einen Blick auf den Code zu werfen, und ihn gegebenenfalls anzupassen. Wenn man eine `load_pcx()` Funktion benötigt, die ein Bild nicht aus einer Datei, sondern aus einem Speicherblock lädt, dann kann man einfach die entsprechende Datei öffnen (in diesem Fall wäre das `allegro/src/pcx.c`) und diese Funktion als Basis für seine eigene nehmen.

Ein weiterer, sehr großer Vorteil ist die aktive Userbasis, die Allegro hat. Auf *http://www.allegro.cc/* findet man neben Neuigkeiten und einem Verzeichnis mit Erweiterungen auch ein Nutzerforum. Gerade als Anfänger ist die schnelle, kompetente und kostenlose Hilfe ein gewaltiger Vorteil. Auf die meisten Fragen bekommt man innerhalb sehr kurzer Zeit eine hilfreiche Antwort.

Spieleprogrammierung leicht gemacht

Die Vielzahl und Qualität der Allegro-Funktionen erlaubt es einem, sich vollständig auf das eigentliche Spiel zu konzentrieren, anstatt sich um Hilfsfunktionalitäten kümmern zu müssen. Dies führt manchmal zu seltsamen Stilblüten, wie zum Beispiel den »Speedhack«-Wettbewerben, in denen innerhalb eines Wochenendes (74 Stunden) ein Spiel erstellt werden muss, das bestimmten Kriterien entspricht. Diese können von dem Genre, einem vorgegebenen Spielprinzip (»Liebe und Frieden«) über ein bestimmtes Gimmick (»Es muss ein Lama im Spiel vorkommen«) bis hin zu technischen Bedingungen (»Es dürfen keine externen Daten geladen werden«) reichen. Am Anfang des Jahres 2003 fand zum ersten Mal der »Blitzhack« statt. Dieser hatte zur Aufgabe, ein Spiel innerhalb von nur 6 Stunden zu bauen. Im Rahmen des Blitzhacks gab es auch ein paar Appetithappen, quasi zum »warm werden«. Eine dieser Übungsaufgaben war eine »MineSweeper« Variante, die in weniger als 2 Stunden geschrieben werden sollte. Und dank Allegro haben das auch die meisten Teilnehmer geschafft. Ein paar waren sogar in deutlich weniger als einer (!) Stunde fertig. Und eine Stunde ist normalerweise die Zeit, die man bei einer DirectX-Applikation ohne Allegro braucht, um seine Arbeitsumgebung komplett einzurichten.

Alternativen

Neben Allegro gibt es noch einige andere Bibliotheken für die Spiele-Entwicklung, die ebenfalls auf mehreren Plattformen laufen. Die Frage welche dieser Bibliotheken besser ist, sollte gar nicht gestellt werden. Die einzige sinnvolle Antwort darauf ist: »Kommt darauf an«. Je nach persönlichen Vorlieben und gestellter Aufgabe kann die eine oder andere Bibliothek besser geeignet sein.

ClanLib

ClanLib ist eine »Medium Level« C++ Bibliothek, die ein Gerüst für eigene Spiele zur Verfügung stellt. Beim Verfassen dieses Buches war die aktuelle stabile ClanLib Version 0.6.

Die von ClanLib unterstützen Plattformen sind Windows und Linux.

ClanLib ist ebenfalls OpenSource, nutzt allerdings eine etwas strengere Lizenz als Allegro. Die LGPL erlaubt es dem Nutzer zwar auch seine Programme ohne Quellcode zu veröffentlichen, schließt dann aber die Verwendung von abgewandelten Quell Code aus. Oder anders gesagt: Man kann zwar ClanLib in einem »Closed Source« Projekt nutzen, darf dann aber nicht abgewandelte Funktionen nutzen. Das oben genannte Beispiel, mit der abgewandelten `load_pcx()` Funktion wäre also in ClanLib nicht möglich.

ClanLib unterstützt auch direkt Netzwerkspiele und das Ogg-Vorbis Soundformat. Bei Allegro sind diese beiden Funktionalitäten derzeit als Add-On verfügbar.

Mehr Informationen zu ClanLib findet sich auf *http://www.clanlib.org/*.

DND3E

Diese Bibliothek geht einen anderen Weg als die restlichen Bibliotheken. Anstatt sich auf Dinge wie Grafik, Sound und Eingabegeräte zu konzentrieren, liegt der Schwerpunkt hier bei der D20-Spielmechanik. Das D20-System ist ein Regelwerk welches die Grundlage der dritten Ausgabe des »Dungeons and Dragons« Rollenspiels bildet. Spiele, die auf D20 beruhen müssen unter der Open Gaming Licence stehen.

Mehr Informationen über DND3E ist auf *http://www.dnd3e.org/* zu finden.

PLib

Eine auf OpenGL basierende Bibliothek für die Spieleentwicklung. Es bietet Funktionen für Benutzeroberflächen, Grafik und Polygonen, Sound, Text und Netzwerkunterstützung.

Alle Plattformen mit OpenGL Support werden unterstützt. Dies schließt natürlich die drei großen Plattformen (Linux, MacOS und Windows) mit ein.

Auch PLib steht unter der LGPL, was auch in diesem Fall wieder eine Nutzung des veränderten PLib Quellcodes verbietet, falls nicht der eigene Quellcode offen gelegt wird.

Simple Direct Media Layer (SDL)

Die SDL versteht sich als Multimedia-Schnittstelle für Spiele, Spiele SDKs, Emulatoren, Demos und Multimedia-Anwendungen. Es bietet einen Low Level Support für Bitmaps, Audioausgabe, Ereignis-Behandlung, Threads und Timer.

Die SDL entstand als eine Bibliothek, die es erlauben sollte, Windows Spiele möglichst einfach auf Linux zu portieren.

Die vollständig unterstützten Plattformen sind Windows und Linux, die Unterstützung für BeOS und MacOS ist noch nicht ganz so weit fortgeschritten.

Die SDL besteht aus einigen Bibliotheken, welche die unterschiedlichen Funktionalitäten zu Verfügung stellen (SDL_Mixer, SDL_Image etc).

Die Lernkurve bei der Simple Direct Media Layer ist eher steil. Bevor man sich an das Erstellen eigener Programme machen kann, ist eine längere Einarbeitungszeit in die API der SDL nötig.

Wie ClanLib steht auch die SDL unter der LGPL, die es »Closed Source« Programmen nicht erlaubt, Teile der Bibliothek in veränderter Form zu nutzen.

Mehr Informationen findet man unter *http://www.libsdl.de/*.

11 Ein erstes Allegro-Programm

Jetzt geht's ans Eingemachte: Wir entwickeln unser erstes Allegro-Programm! Ich gehe davon aus, dass Allegro und der GCC bereits installiert sind. Sollte dies nicht der Fall sein, dann wäre jetzt ein guter Zeitpunkt zum Installieren. Genaue Installationsanweisungen finden Sie im Anhang.

```
#include <allegro.h>
int main(int argc, char **argv) {
    allegro_init();
    allegro_message("Hallo Welt!");
} END_OF_MAIN()
```

Dieses Programm lässt sich auf einem Windows System wie folgt kompilieren:

```
g++ prog1.cpp -o prog1.exe -lalleg
```

Auf einem Linux System sieht diese Zeile so aus:

```
g++ prog1.cpp -o prog1 'allegro-config --libs'
```

Der Unterschied zwischen den Plattformen liegt darin, dass die benötigten Bibliotheken auf einem Linux System variieren können. Allegro legt deswegen ein Script an, welches die korrekten Bibliotheken ermitteln kann. Ein Hinweis, um Schwierigkeiten zu vermeiden: Die einzelnen »Gänsefüßchen« sind Backticks, keine normalen Apostrophe. Man erhält sie über die Taste zwischen dem [ß] und [⇦].

Wenn alles glatt ging, sollte eine Nachrichtenbox mit dem Text »Hallo Welt« erscheinen. Je nach Plattform könnte es auch sein, dass der Text nur auf der Konsole ausgegeben wird.

Wichtig ist, dass ein Allegro-Programm immer eine `main()` Funktion hat. Auch unter Windows und anderen Plattformen, auf denen eigentlich keine `main()` Funktion vorhanden ist. Wichtig ist außerdem, dass das Ende von `main()` durch ein `END_OF_MAIN()` gekennzeichnet ist. Dieses Makro erledigt die Konvertierung in die entsprechende Plattform-spezifische Funktion.

Nun ist dies bisher noch keine berauschende Leistung. Also wagen wir uns nun an etwas Schwierigeres heran: Das Setzen eines Grafikmodus.

Grafikmodi

Grob gesagt haben wir zwei Möglichkeiten, unser Programm laufen zu lassen: im Fenster- oder im Vollbild- Modus. Beide Modi haben ihre Vor- und Nachteile. Der Fenstermodus ist weniger aufdringlich. Ein kleines Spiel wie Minesweeper kann eventuell im Fenster mehr Erfolg haben als wenn es im Vollbild-Modus läuft. Auch hat der Fenstermodus für Entwickler den Vorteil, dass man noch immer die Konsole und damit mögliche Debug-Ausgaben sehen kann. Allerdings ist er auch meist etwas langsamer als der Vollbild Modus. Der Grund hierfür liegt in der Tatsache begründet, dass ein Fenster immer mit der gleichen Anzahl an Farben läuft wie der aktuelle Desktop. Sollte der Nutzer also 24 Bit Farbtiefe eingestellt haben, das Spiel jedoch mit 16 oder 32 Bit Farbtiefe laufen, dann muss bei jedem Update des Fensters zwischen der Anzahl der Farben im Spiel und der Anzahl der Farben auf dem Bildschirm umgerechnet werden. Zwar ist Allegro recht flott bei dieser Aufgabe, jedoch ist ein kleiner Geschwindigkeitsverlust nicht vermeidbar.

Die beiden Funktionen, die wir für das Setzen des Grafikmodus brauchen, sind:

```
void set_color_depth(int depth);
```

Diese Funktion setzt die verwendete Farbtiefe. Sie sollte stets vor dem Aufruf von `set_gfx_mode()` aufgerufen werden. Der Parameter `depth` gibt die Anzahl der Bits an, die für einen Pixel benötigt werden. Gültige Werte sind 8, 15, 16, 24 und 32.

```
int set_gfx_mode(int card, int w, int h, int v_w, int v_h);
```

Diese Funktion macht die eigentliche Arbeit und wechselt in den angegebenen Modus. Die übergebenen Parameter haben folgende Aufgabe:

- `card`: Der verwendete Grafiktreiber. Als Faustregel sollte man entweder `GFX_AUTODETECT`, `GFX_AUTODETECT_FULLSCREEN` oder `GFX_AUTODETECT_WINDOWED` verwenden. Wenn es egal ist, ob Vollbild oder Fenster gewählt wird, dann ist `GFX_AUTODETECT` die richtige Wahl. Allegro wird sich dann automatisch für den schnelleren der beiden Modi entscheiden. Wenn man jedoch explizit im Vollbild-Modus laufen möchte, dann ist `GFX_AUTODETECT_FULLSCREEN` die richtige Wahl. Der Modus `GFX_AUTODETECT_WINDOWED` entspricht, man mag es nicht für möglich halten, dem Fenster-Modus.

- ✔ `w, h`: Die gewünschte Auflösung. Normale Werte für Vollbildanwendungen sind (640, 480), (800, 600) und (1024, 768).
- ✔ `v_w, v_h`: Erlauben es eine virtuelle Auflösung anzugeben. Diese Funktion wird heutzutage leider nicht mehr von vielen Grafikkarten unterstützt. Ich empfehle aus diesem Grund, sie auf den Wert 0,0 zu setzen.

Diese Funktion liefert einen Wert größer oder gleich Null zurück wenn sie erfolgreich ist, ein Wert kleiner als Null zeigt einen Fehler an.

Sobald der Grafikmodus erfolgreich gesetzt wurde, können wir Grafiken anzeigen, in dem wir auf die globale Variable `screen` zugreifen. Diese ist vom Typ »Zeiger auf eine `BITMAP` Struktur«. Wir können einzelne Punkte einer `BITMAP` ändern, indem wir die Funktion `putpixel()` benutzen. Eine Warnung vorweg: `putpixel()` ist eine der langsamsten Funktionen, insbesondere wenn man mit ihr direkt auf `screen` zugreift. Es ist jedoch auch eine der grundlegenden Funktionen, und aus diesem Grund wird sie wohl immer noch häufig zu Demonstrationszwecken verwendet.

```
void putpixel(BITAMP *bmp, int x, int y, int color);
```

Setzt einen Pixel von `bmp` an der Position `x,y` auf die Farbe `color`.

Wir benutzen `putpixel()`, um ein farbiges Rechteck auf den Schirm zu bringen:

```
#include <allegro.h>
int main(int argc, char **argv) {
    allegro_init();
    /* Wir wollen weiter unten auf einen Tastendruck
       warten, aus diesem Grund müssen wir erst mal
       den Keyboard Handler installieren.
     */
    install_keyboard();

    set_color_depth(16);
    if (set_gfx_mode(GFX_AUTODETECT, 640, 480, 0, 0) < 0) {
        allegro_message("Unable to set graphic mode!");
        exit(0);

    }


    for (int r = 0; r < 256; r++) {
```

```
        for (int g = 0; g < 256; g++) {
            /* Wir benutzen r und g auch als
               Bildschirm Koordinaten.
             */
            putpixel(screen, r,g, makecol(r,g,0));
        }
    }
    /* Auf Tastendruck warten */
    while (!keypressed());
} END_OF_MAIN();
```

Dieses Programm finden Sie unter dem Namen prog2.cpp auf der beiliegenden CD im Verzeichnis für dieses Kapitel.

Wenn Ihre Grafikkarte einen 16-Bit-Modus unterstützt, so wird dieses Programm mit hoher Wahrscheinlichkeit im Vollbildmodus ausgeführt. In diesem Fall werden Sie auch den Eindruck haben, dass das Rechteck sofort erscheint, weil die Zeit, die der Monitor braucht, um sich an die neue Auflösung anzupassen, in der Regel länger ist als die Zeit, die das Programm braucht, um das Rechteck zu malen.

Sollte Ihre Grafikkarte jedoch als Hi-Color-Modus 15-Bit-Farbtiefe verwenden, dann können Sie sich am spaltenweisen Aufbau des Rechtecks erfreuen. Wie gesagt: `putpixel()` ist wirklich langsam. Und auch Besitzer einer Grafikkarte die den Modus unterstützt können sich einen Eindruck über die quälende Geschwindigkeit verschaffen, indem Sie `GFX_AUTODETECT` durch `GFX_AUTODETECT_WINDOWED` ersetzen.

Nun gibt es einige kleine Probleme mit diesem Programm. Angenommen, wir würden gerne sicher stellen, dass wir auf jeden Fall im Vollbildmodus laufen. Für diese Gewissheit müssen wir leider die Gewissheit aufgeben mit 16-Bit-Farbtiefe zu laufen. Wir versuchen als erstes 16-Bit-Vollbild, dann 15-Bit-Vollbild und falls auch dies nicht funktionieren sollte 16 Bit im Fenster. Aber die Chancen sind wirklich mehr als gut, dass jede Grafikkarte zumindest einen dieser Modi unterstützt.

Eine zweite Verbesserung sollte uns einen deutlichen Geschwindigkeitsvorteil bringen. Für jeden Zugriff auf den Bildschirm ist es nötig, den Speicherbereich zu sperren. In unserer bisherigen Version verlassen wir uns darauf, dass `putpixel()` für uns den Bildschirm sperrt, dann den Pixel setzt, und dann den Schirm wieder freigibt. Wenn man nun noch weiß, dass das Sperren des Schirms verhältnismäßig lange braucht, wird das Problem deutlich.

Aus diesem Grund ist es immer eine gute Idee, vor dem Aufruf mehrerer Funktionen die auf den Bildschirmspeicher zugreifen, diesen zu sperren.

```
void acquire_screen();
```

Sperrt den Bildschirmspeicher.

```
void release_screen();
```

Gibt den gesperrten Schirm wieder frei.

In `prog2.cpp` kann das simple Einfügen von `acquire_screen()` vor der Schleife und `release_screen()` nach der Schleife einen gewaltigen Unterschied ausmachen.

Nun aber, ohne weitere Vorrede: `Prog3.cpp`

```
#include <allegro.h>
int set_graphic_mode() {
    set_color_depth(16);
    if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0) >= 0)
{
        return 1;
    }
    set_color_depth(15);
    if (set_gfx_mode(GFX_AUTODETECT_FULLSCREEN, 640, 480, 0, 0) >= 0)
{
        return 1;
    }
    set_color_depth(16);
    if (set_gfx_mode(GFX_AUTODETECT_WINDOWED, 640, 480, 0, 0) >= 0) {
        return 1;
    }
    return 0;
}
int main(int argc, char **argv) {
    allegro_init();
    install_keyboard();

    if (!set_graphic_mode()) {
        allegro_message("Unable to set graphic mode!");
        exit(0);
    }
    acquire_screen();
    for (int r = 0; r < 256; r++) {
        for (int g = 0; g < 256; g++) {
            putpixel(screen, r,g, makecol(r,g,0));
        }
```

```
    }
    release_screen();

    while (!keypressed());
} END_OF_MAIN();
```

Das Setzen des Grafikmodus wurde in eine eigene Methode ausgelagert, die 0 zurückliefert, wenn kein Modus gesetzt werden kann, und einen Wert ungleich 0, wenn ein Modus erfolgreich gesetzt wurde.

Allerdings ist das Setzen einzelner Pixel zwar sehr grundlegend, aber auch mindestens genauso langweilig. Der nächste Schritt macht alles etwas interessanter: das Laden und Anzeigen von Bildern.

Bilder laden und anzeigen

Bilder sind das grafische Herzstück eines jeden 2D-Spiels. Letztendlich reduzieren sich die meisten Spiele auf das clevere Anzeigen von Bitmaps. Doch bevor man die Grafiken auf den Schirm bringen kann, muss man sie erst einmal laden.

Glücklicherweise ist dies mit Allegro recht einfach.

```
BITMAP *load_bitmap(const char*filename, RGB* pal);
```

Lädt ein Bild in einem der unterstützten Dateiformaten (BMP, LBM, PCX, TGA). Gibt einen Zeiger auf die geladene `BITMAP` zurück oder `NULL` wenn das Bild nicht geladen werden konnte.

- ✔ `filename`: Name und gegebenenfalls Pfad der zu ladenden Datei. Das Dateiformat der Grafik wird Anhand der Erweiterung ermittelt.
- ✔ `pal`: Zeiger auf ein Array von 256 `RGB` Werten. In dieses wird die Palette des geladenen Bildes (wenn vorhanden) geladen. Da wir uns im 16-Bit-Farbmodus befinden, können wir diesen Parameter auf `NULL` setzen.

Bilder sollten immer erst nach dem Setzen des Grafikmodus geladen werden, da Allegro das Bild standardmäßig beim Laden in ein zum Bildschirm kompatibles Format konvertiert.

Um das Bild dann schließlich anzuzeigen, wird eine weitere Funktion benötigt:

```
void blit(
    BITMAP *source, BITMAP *dest,
    int source_x, int source_y,
    int dest_x, int dest_y,
    int width, int height
);
```

- ✔ source: Die BITMAP, die das anzuzeigende Bild enthält.
- ✔ dest: Die BITMAP, auf der das Bild gezeigt werden soll (zum Beispiel screen).
- ✔ source_x, source_y: Die obere linke Ecke des Bereichs, der angezeigt werden soll (normalerweise 0,0).
- ✔ dest_x, dest_y: Die Position, an der das Bild angezeigt werden soll.
- ✔ width, height: Größe des anzuzeigenden Bereichs.

Die reine Anzahl an Parametern ist am Anfang etwas erschreckend, aber in den meisten Fällen wird man source_x und source_y auf 0 setzen und width und height auf die Größe der anzuzeigenden Bitmap.

Ändern Sie die main() Funktion folgendermaßen ab und speichern Sie den Quellcode dann als prog4.cpp.

```
int main(int argc, char **argv) {
    allegro_init();
    install_keyboard();

    if (!set_graphic_mode()) {
        allegro_message("Unable to set graphic mode!");
        exit(0);
    }
    // Laden des Bildes
    BITMAP *logo = load_bitmap("allegro.tga", NULL);

    acquire_screen();
    clear(screen);
    blit(logo, screen,
         0, 0,
        (SCREEN_W - logo->w)/2, // zentriert
        (SCREEN_H - logo->h)/2, // zentriert
        logo->w, logo->h);
    release_screen();

    while (!keypressed());
} END_OF_MAIN();
```

Kompiliert wird auch dieses File mit:

```
g++ -o prog4.exe prog4.cpp -lalleg
```

Stellen Sie sicher, dass sich `allegro.tga` im gleichen Verzeichnis wie die ausführbare Datei befindet.

Dieses Bild finden Sie zusammen mit der Version des Quellcodes im Verzeichnis dieses Kapitels auf der Buch-CD.

Nach dem Laden des Bildes zeigen wir dieses zentriert auf dem Schirm an. Wenn alles funktioniert wie erwartet, dann sieht das Ergebnis so aus:

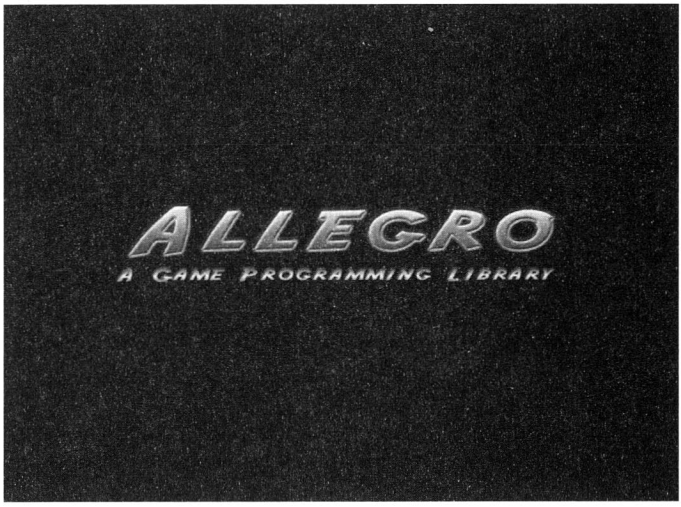


Abbildung 11.1: Die Ausgabe von prog4.exe

Nachdem wir nun Bilder laden und anzeigen können, gehen wir einen Schritt weiter und setzen das Bild in Bewegung.

Bewegung

Erstellen Sie eine Kopie des Quellcodes von `prog4` und speichern Sie diese unter `prog5.cpp`.

Die entscheidende Zeile im Quellcode ist der Aufruf von `blit()`. Wenn wir in dieser Zeile die feste Position durch Variablen ersetzen, dann können wir die Position des Bildes beliebig ändern.

```
blit(logo, screen, 0, 0, x, y, logo->w, logo->h);
```

Wenn wir nun x und y ändern und dann das Bild noch mal anzeigen, entsteht der Eindruck einer flüssigen Bewegung – jedenfalls wenn wir schnell genug sind.

```
int x=0;
int y=0;
while (!keypressed()) {
    clear(screen);
    blit(logo, screen, 0, 0, x, y, logo->w, logo->h);
    x += 1;
    y += 1;
}
```

Jetzt bewegt sich das Logo von oben links nach unten rechts. Und dann aus dem Bildschirm hinaus. Da die Position sich immer weiter erhöht, unser Bildschirm aber nur 640 Pixel breit und 480 Pixel hoch ist (bei der derzeitigen Auflösung), müssen wir Vorkehrungen treffen, damit das Logo nicht einfach so verschwindet.

Ein gern benutzter Effekt ist es, die Grafik von den Kanten des Bildschirms abprallen zu lassen. Quasi also, als ob das Logo der Ball in einer Partie »Pong« wäre. Trifft unsere Grafik eine Wand, dann ändert sie die Richtung. Bewegt sie sich nach unten und stößt gegen den unteren Rahmen, dann bewegt sie sich ab da nach oben. Prallt sie gegen den rechten Rahmen, so bewegt sie sich ab diesem Zeitpunkt nach links, beim oberen Rahmen dementsprechend nach unten und beim linken Rahmen nach rechts.

Für uns bedeutet das, dass sich nicht nur die Position des Logos ändern kann, sondern auch die Richtung, in die es sich bewegt. Also definieren wir zwei weitere Variablen, welche die Richtungsänderung speichern. Eine Änderung bezeichnet man auch als Delta-Wert. Aus diesem Grund werden die Variablen, die eine Änderung der x- und y-Position verursachen gerne dx und dy genannt.

Kollisionsabfrage mit dem Rahmen

Um nun die Richtung zu ändern, müssen wir feststellen, ob eine Kollision mit dem Rahmen stattfinden würde. Widmen wir uns zuerst der Kollision mit dem linken und rechten Rand:

```
if (x + dx < 0) {
    // Kollision mit linken Rand
    dx*= -1;
} else if (x + logo->w + dx >= SCREEN:W) {
    // rechter Rand
    dx *= -1;
}
```

Wir überprüfen erst, ob die x-Position nach der Bewegung kleiner als 0 ist. In diesem Fall wären wir außerhalb des Bildschirmrahmens. Um dies zu vermeiden, multiplizieren wir den `dx` Wert mit –1. Anstatt nach links bewegen wir uns nun nach rechts.

Die Abfrage mit dem rechten Rand ist etwas komplexer. Da wir den rechten Rand des Logos überprüfen müssen, ist es notwendig zu der x-Position auch noch die Breite des Logos zu addieren. Ist das Ergebnis größer oder gleich der Breite des Schirms, findet eine Kollision statt. Auch in diesem Fall multiplizieren wir `dx` mit –1, um die Richtung umzukehren.

Man könnte die beiden Abfragen zu einer zusammenfassen, da ja in beiden Fällen der gleiche Code ausgeführt wird. Da ich aber vorhabe, mit diesem Beispiel noch etwas weiterzuarbeiten, und sich der ausgeführte Code noch ändern wird, ist es zweckmäßiger, in diesem Fall 2 `if`-Abfragen zu nutzen.

Der Code für die y-Position entspricht eigentlich dem für die x-Position.

Die komplette Schleife sieht nun so aus:

```
int x = 0;
int y = 0;
int dx = 1;
int dy = 1;
while (!keypressed()) {

    // Linker und rechter Rand
    if (x + dx < 0) {
        dx *= -1;
    } else if (x + logo->w >= SCREEN_W) {
        dx *= -1;
```

```
    }
    // Oberer und unterer Rand
    if (y + dy < 0) {
        dy *= -1;
    } else if (y + logo->h >= SCREEN_H) {
        dy *= -1;
    }
    x+= dx;
    y+= dy;

    acquire_screen();
    clear(screen);
    blit( logo, screen, 0, 0, x, y, logo->w, logo->h);
    release_screen();
}
```

Das komplette Programm ist unter dem Namen prog5.cpp auf der CD zu finden.

Nun sieht das ja gar nicht so schlecht aus, hat jedoch den Nachteil, dass es recht stark flackert. Um den Grund für das Flackern zu begreifen, muss man verstehen, wie ein Monitor prinzipiell funktioniert.

Der Rasterstrahl des Monitors bewegt sich von links nach rechts und von oben nach unten über den Bildschirm. Wo er den Schirm trifft, wird ein einzelner Bildpunkt angezeigt. Am Ende einer jeden Zeile schaltet er sich selbst kurz ab und saust zum Anfang der nächsten Zeile. Dieses »zurücksausen« am Ende der Zeile nennt man den horizontalen Strahlrücklauf, oder auf Englisch »horizontal retrace«. Ist der Strahl am Ende der letzen Zeile angekommen, hat er einen noch weiteren Weg vor sich: zurück nach oben, zum ersten Pixel des Schirms. Dies bezeichnet man als »vertikalen Strahlrücklauf«, auf Englisch »vertical retrace«.

Durch die Trägheit des Auges bekommen wir von dem Aufbau nichts mit, und sehen ein stabiles Bild.

Wenn wir nun den Inhalt des Bildschirmspeichers (`screen`) ändern, so sehen wir diese Änderung erst, wenn der Rasterstrahl das nächste Mal an dieser Stelle ist.

Das Flackern rührt nun daher, dass im Augenblick wenn wir den Bildschirm löschen (`clear( screen );`), der Rasterstrahl im Bereich des Logos ist. Dies führt dazu, dass dieser Bereich für einen sehr kurzen Moment schwarz dargestellt wird, bevor beim nächsten Mal, wenn der Ra-

sterstrahl an dieser Stelle ist, der korrekte Wert im Bildschirmspeicher steht.

Die Lösung für dieses Problem ist recht einfach: Anstatt die Pixel zweimal zu setzen (erst auf schwarz und dann auf die Farbe des jeweiligen Pixels im Logo), setzen wir jeden Bildschirmpunkt nur einmal.

Doublebuffering

Um das Flackern zu vermeiden, führen wir jede Änderung zuerst in einem gesonderten, nicht sichtbaren Bereich durch. Diesen Bereich nennt man meist offscreen Buffer (da er nicht auf dem Bildschirm sichtbar ist) oder double Buffer (da wir zwei Speicherbereiche haben: den Schirm selbst und den nicht sichtbaren).

Wenn nun alle Änderungen auf dem nicht sichtbaren Buffer erledigt sind, wird dieser in einem Rutsch in den Bildschirmspeicher kopiert.

Genug der Theorie, ran an die Praxis. Zuerst benötigen wir einen zweiten Buffer. Damit wir wie bisher damit arbeiten können, muss dieser natürlich auch eine `BITMAP` sein. Allegro hat die Funktion `create_bitmap()`, die uns eine `BITMAP` nahezu beliebiger Größe (innerhalb eines vernünftigen Rahmens) erstellt.

```
BITMAP *create_bitmap(int width, int height);
```

Diese Funktion erzeugt entweder eine Bitmap der angegebenen Größe oder liefert bei einem Fehler `NULL` zurück.

Wenn wir eine Bitmap mit der gleichen Größe des Bildschirms erzeugen und dann alle Funktionen, die bisher auf den Bildschirmspeicher zugegriffen haben, so ändern, dass sie auf diese Bitmap zugreifen, dann haben wir es geschafft.

```
BITMAP *doublebuffer = create_bitmap(SCREEN_W, SCREEN_H);
// … hier bleibt alles wie bisher
while (!keypressed()) {
    // an der Kollisionsabfrage ändert sich nichts
    acquire_bitmap(doublebuffer);
    clear(doublebuffer);
    blit( logo, doublebuffer,
          0, 0, x, y, logo->w, logo->h);
    release_bitmap(doublebuffer);

    blit(doublebuffer, screen,
         0, 0, 0, 0, SCREEN_W, SCREEN_H);
}
```

Der komplette Quellcode findet sich wieder auf der CD, der Name der Datei ist prog6.cpp

Sound

Die Musik und die Soundeffekte in einem Spiel machen einen großen Teil der Stimmung aus. Gute Soundeffekte können aus einem Spiel ein großartiges machen. Stellen wir uns folgende Szene vor: Das Sprite des Helden läuft durch ein Dungeon. Es ist recht dunkel, nur die Fackeln links und rechts erhellen den Gang etwas.

Schließen Sie jetzt bitte die Augen und stellen Sie sich diese Szene vor. In völliger Stille.

Gut. Jetzt stellen Sie sich bei jedem Schritt hallende Gehgeräusche vor. Haben Sie das? Und jetzt noch das Geräusch eines pochenden Herzschlags. Das Herz schlägt immer lauter und schneller je näher der Held sich der Tür am Ende des Ganges nähert ...

Falls Sie immer noch Zweifel an der Bedeutung der Soundeffekte haben, schauen Sie sich einmal eine Actionszene ohne Sound an. Einfach den Ton abstellen. Und schon verliert auch die gewaltigste Action deutlich an Dynamik. Dann machen Sie die Gegenprobe: Ton an und Augen zu. Und schon fängt Ihre Fantasie an, Bilder in Ihrem Geiste zu malen, die zu den Geräuschen passen. Durch stimmige Soundeffekte erhält Ihre Spielwelt mehr Tiefe, wird der Spieler stärker in das Spiel eingebracht.

Und das Beste ist: All dies können Sie mit nur ein paar Zeilen Code mehr erreichen.

Initialisierung

Bevor man Sound und Musik nutzen kann, muss das entsprechende Subsystem von Allegro erst initialisiert werden. Dies erreicht man durch einen Aufruf von

```
int install_sound(int digi, int midi, char* reserved);
```

Installiert das Soundsystem. Normalerweise werden Sie für `digi` immer `DIGI_AUTODETECT` und für `midi` immer `MIDI_AUTODETECT` verwenden. Der Parameter `reserved` wird derzeit nicht verwendet und sollte auf `NULL`

gesetzt werden. Sollte ein Fehler aufgetreten sein, so ist der Rückgabewert –1, bei Erfolg wird 0 zurückgegeben.

Der nächste Schritt ist das Einlesen eines Soundfiles. Allegro unterstützt WAV- und VOC-Dateien in Mono und Stereo. Es stehen Erweiterungen zur Verfügung, die das Laden von Ogg-Vorbis und MP3-Dateien erlauben.

Um eine Sounddatei zu laden, nutzen Sie folgende Funktion:

```
SAMPLE *load_sample(const char *filename);
```

Wird die Datei gefunden und ist sie ein einem unterstützen Format, so liefert ein Aufruf dieser Funktion einen Zeiger auf ein `SAMPLE` zurück. Gibt es ein Problem, ist der Rückgabewert `NULL`.

Sobald Sie ein `SAMPLE` geladen haben, können Sie es mittels

```
int play_sample(
    const SAMPLE *spl,
    int vol, int pan, int freq, int loop);
```

abspielen.

Der erste Parameter ist ein Zeiger auf eine gültige `SAMPLE` Struktur.

Der Parameter `vol` gibt die Lautstärke relativ zur derzeitig gesetzten Maximallautstärke an. Gültige Werte liegen zwischen 0 (kein Sound) bis hin zu 255 (maximale Lautstärke).

Der Parameter `pan` erlaubt es die Stereoposition zu ändern, an der der Sound abgespielt wird. Bei einem Wert von 128 wird der Sound gleichmäßig auf beiden Kanälen abgespielt. Ein Wert von 0 bedeutet, dass der Sound nur auf dem linken Lautsprecher abgespielt wird. Ein Wert von 255 bedeutet, dass der Sound nur rechts abgespielt wird.

Mit `freq` kann man die Geschwindigkeit beeinflussen, mit der der Sound abgespielt wird. Ein Wert von 1000 bedeutet normale Geschwindigkeit. Ein kleinerer Wert verlangsamt den Sound, ein größerer Wert sorgt dafür, dass der Sound schneller gespielt wird. Dabei sorgt eine Verdoppelung des Wertes auch für eine Verdoppelung der Geschwindigkeit. 4000 wäre also ein 4 mal so schneller Sound, 500 ein Sound, der mit der halben Geschwindigkeit abgespielt wird.

Der letzte Parameter, `loop`, bestimmt, ob der Sound nur einmal abgespielt wird oder ob der Sound sich immer wieder wiederholt. Übergeben Sie hier 0, um den Sound nur einmal abzuspielen. Ein Wert, der von 0

verschieden ist, sorgt dafür, dass der Sound wieder und wieder von vorne abgespielt wird. In diesem Sound muss das Sample explizit via `stop_sample()` gestoppt werden.

Der Standardaufruf für `play_sample()` sieht also so aus:

```
// soundfx mit voller Lautstärke zentriert abspielen
play_sample(soundfx, 255, 129, 100, 0);
```

Konstante Geschwindigkeit

Die Chance zwei Computer zu finden, die ein Spiel mit genau der gleichen Geschwindigkeit ausführen, ist verschwindend gering. Computer kommen heutzutage in vielfältigen Konfigurationen. Am unteren Ende stehen zur Zeit Rechner mit einem Pentium II und 200 MHz, am oberen Ende der Pentium IV mit Takten nahe an 3 GHz. Grafikkarten der neueren Rechner haben zum Teil die gleiche Menge an Speicher, den die älteren als RAM haben. Und selbst wenn man zwei Rechner, die mit der gleichen Hardware ausgestattet sind vergleicht, können Unterschiede in der Software (Treiberversionen, Systemkonfiguration) zu Unterschieden führen.

Normalerweise ist dies kein großes Problem. Dass ältere Rechner langsamer sind, wird weitestgehend akzeptiert. Und man kauft sich ja schließlich einen neuen Rechner, damit die Programme schneller ausgeführt werden. Wenn jedoch ein Spiel schneller ausgeführt wird, dann kann es zu Problemen kommen. Gehen wir einmal davon aus, dass dieser Code benutzt wird, um ein Geschoss über den Bildschirm zu bewegen:

```
bullet.x = 0;
while (bullet.x < SCREEN_W) {
    bullet.x++;
    // display bullet...
}
```

Sieht ja auf den ersten Blick ganz gut aus. Das Problem ist nur, dass man vorher nicht wissen kann, wie lange es dauert, bis die Kugel auf der anderen Seite des Schirms ankommt. Auf einem langsamen Computer könnte es mehrere Sekunden dauern, bis die Kugel ankommt, auf einem schnellen Rechner kann man die Kugel vielleicht nicht einmal sehen, da sie wieder verschwunden ist, bevor man sie überhaupt gesehen hat. Was dem Spieler natürlich auch keine Zeit gibt zu reagieren.

Und wenn man nun nicht nur die Sprites, sondern auch den Spieler auf ähnliche Weise bewegt, dann könnte dies in etwa so aussehen:

```
while (!gameOver) {
    if (key[KEY_LEFT] {
        player.x --;
   } else if (key[KEY_RIGHT]) {
       player.x++;
   }
   // display player
}
```

Auf einem zu langsamen Rechner könnte es nahezu ewig dauern, bis der Spieler von einem Ende des Schirms auf den anderen kommt. Auf einem zu schnellen Rechner könnte der Charakter in einen Gegner laufen, bevor der Spieler Zeit hat zu reagieren, oder er stürzt von einer Plattform, wird von einem Hammer zerquetscht oder ... na ja, ich denke es ist klar geworden, dass dem Sprite ein finsteres Ende droht.

Wir müssen also einen Weg finden dafür zu sorgen, dass das Spiel auf jedem Rechner gleich schnell läuft. Eine Methode wäre, einen Faktor einzubauen und dann alle Bewegungswerte mit diesem Faktor zu multiplizieren. Auf schnellen Rechnern ist dieser Faktor klein (sagen wir mal 0.02) auf langsamen Rechnern dann deutlich größer (zum Beispiel 2.5). In dem Beispiel mit dem Geschoss könnte dies in etwa so aussehen:

```
float factor = calcSpeedFactor();
bullet.x = 0;
bullet.speed = 1.0;
while (bullet.x < SCREEN_WIDTH) {
    bullet.x += bullet.speed * factor;
    // display bullet...
}
```

Dies löst das Problem mit der unterschiedlichen Geschwindigkeit. Allerdings kommen nun ein paar neue Probleme dazu. Alle Positionen sollten nun Gleitkommazahlen (floats) sein. Und die sind besonders auf älteren Rechnern eine weitere Bremse. Außerdem muss man auch erst einmal den Geschwindigkeitsfaktor berechnen. Dies könnte in etwa so aussehen:

```
long time = currentTime();
// Hier wird irgendetwas gemacht, das Spiel typisch
// ist und länger dauert.
// Das anzeigen von Grafiken und das Berechnen
// einige Werte wären ein gutes Beispiel
```

```
long time2 = currentTime();
long diff = time2 - time;
factor = (float) NORMAL_DIFF / diff;
```

Dies sieht auf den ersten Blick auf nach einer guten Idee aus. Und einige kommerzielle Spiele der Vergangenheit scheinen auch eine solche Methode benutzt zu haben. Und wenn ein älteres Spiel direkt nach dem Start mit einem »Division by Zero Error« abbricht, dann hat man ein solches in den Händen.

Auf schnellen Rechnern kann die gestoppte Operation so schnell durchgeführt werden, dass die verstrichene Zeit nicht gemessen werden kann. Und dann ist `diff` 0. Was dazu führt, dass bei der Berechnung des Faktors durch 0 geteilt wird. Das Ergebnis ist ein Programmabbruch.

Zum Glück gibt es eine recht einfache Lösung für unser Problem.

Konstante logische Frame Rate

Gehen wir einmal davon aus, dass wir uns die Frame Rate des Spiels beliebig aussuchen können. Wie viele Bilder pro Sekunde (Frames per Second, fps) brauchen wir denn eigentlich? Natürlich ist man am Anfang geneigt zu sagen: »Möglichst hoch.« Schließlich ist eine hohe Frame Rate ja auch das Maß aller Dinge. Allerdings machen diese vielen Frames nur dann einen Sinn, wenn wir auch genug Grafiken haben. Eine »normale« Laufanimation hat zwischen 5 und 16 verschiedene Bilder. Da aber ein Schritt immer gleich lang dauert, bedeutet dies, dass einige Phasen der Animation länger angezeigt werden. In diesem Fall hat also die höhere Bildrate keinen Einfluss auf die Qualität der Animation. Hier ist eine kurze Übersicht über die gängigen Bildraten:

Trickfilmserie	15 Bilder pro Sekunde
Fernsehen	25 Bilder pro Sekunde
Spielkonsolen	50 Bilder pro Sekunde (PAL) 60 Bilder pro Sekunde (NTSC)

Tabelle 11.1: Gängige Bildraten

Trickfilmserien werden mit 15 Bildern pro Sekunde gemalt. Allerdings besteht eine Sekunde Fernsehen aus 25 Einzelbildern. Was wieder dazu führt, dass einige Animationsphasen länger zu sehen sind als andere. Nun sehen diese Trickfilme aber wirklich recht fließend aus. Können wir uns also mit 15 Bildern pro Sekunde begnügen? Was die Animationen angeht, sind 15 Bilder pro Sekunde womöglich wirklich ausreichend. Aber bei der Bewegung von Objekten oder des Bildschirmausschnitts ist eine höhere Frame Rate ein großer Vorteil. Wenn unser Spiel mit 15 Bildern / Sekunde läuft, und wir ein Sprite in einer Sekunde 100 Pixel weit bewegen wollen bedeutet dies, dass sich das Objekt in jedem Frame um 6.6 Pixel bewegen muss. Und 6 Pixel sind eine recht große Distanz. Auch wird die Bewegung nicht gleichförmig aussehen, da Rundungsfehler die Anzeige beeinträchtigen.

Wenn das Spiel statt mit 15 mit 30 Frames pro Sekunde läuft, dann sind es nur noch 3.3 Pixel pro Bild. Auch die Abweichung wird deutlich geringer. Und bei 100 Bildern pro Sekunde bewegt sich das Objekt schließlich fließend von Pixel zu Pixel.

Daraus folgt, dass die Auflösung des Spieles einen großen Einfluss auf die bevorzugte Bildwiederholrate hat. Bei einer Auflösung von 320x240 Pixeln ist eine Frame Rate von 30 Bildern pro Sekunde ausreichend. Geht man mit der Auflösung höher, dann sollte auch die Frame Rate angepasst werden.

Auflösung	Bildwiederholrate
320x240	25 Bilder pro Sekunde
640x480	50 Bilder pro Sekunde
800x600	60 Bilder pro Sekunde
1024x768 (und höher)	80 Bilder pro Sekunde

Tabelle 11.2: Empfohlene Bildraten

Ab 80 Bilder pro Sekunde wird das Auge so perfekt getäuscht, dass die meisten Menschen keine Verbesserung der Bildqualität feststellen können, auch wenn man die Anzahl der dargestellten Bilder weiter erhöht.

Angenommen wir wählen nun eine Wiederholrate von 60 Bildern pro Sekunde für das Spiel. Wie garantieren wir nun eine gleichbleibende Geschwindigkeit auf allen Rechnern?

Die Antwort ist verblüffend einfach. Der zeitaufwendige Teil eines Spieles ist in der Regel nicht die Ablauflogik, sondern das Darstellen der Grafiken auf dem Bildschirm. Wenn wir nun die Logik von der Darstellung trennen, können wir davon ausgehen, dass wir die Logik immer mit der vorgegebenen Anzahl an Frames rechnen lassen können. Dies garantiert eine konstante Geschwindigkeit aller Objekte auf dem Schirm. Auf langsamen Rechnern werden allerdings nicht alle diese Frames wirklich angezeigt werden.

Um nun diese »logische Frame Rate« zu erreichen, brauchen wir nur einen Timer, der eine Variable entsprechend erhöht. Bei 60fps muss der Timer also 60 mal pro Sekunde aufgerufen werden.

```
volatile int timerCounter = 0;
static void timerCounterUpdater() {
    timerCounter++;
}
END_OF_STATIC_FUNCTION(timerCounterUpdater);

// in der initialisierungs- Methode

// Funktion und Variable locken
LOCK_FUNCTION(timerCounterUpdater);
LOCK_VARIABLE(timerCounter);

// 60 Aufrufe pro Sekunde
install_int_ex(timerCounterUpdater, BPS_TO_TIMER(60));
```

Nun können wir unseren Main Loop so ändern, dass die Spiellogik 60 mal pro Sekunde ausgeführt wird.

```
if (timerCounter > 0) {
    do {
        /* Hier kommt die eigentliche Logik */

        /* Buchführung, um die 60fps zu sichern */
        cur.logic++;
        timerCounter--;
    } while (timerCounter > 0);
    needsRefresh = TRUE;
}
```

```
if (needsRefresh) {
    /* Grafiken anzeigen*/
    blit( ... );
    needsRefresh = FALSE;
}
```

Wenn `timerCounter` einen Wert größer als 0 hat, dann führen wir die Logik so lange aus, bis der Timer wieder bei 0 ist. Dadurch können wir auch bei längeren Verzögerungen die logische Wiederholrate beibehalten. Eine kleine Fehlerquelle steckt allerdings in diesem Code. Sollte die Logik länger als 1/60tel Sekunde dauern, hängt das Programm in einer Endlosschleife.

Zwar sollte dies nicht vorkommen, aber Vorsicht ist besser als sich am Ende mit Bugreports herumschlagen zu müssen. Fügen wir also einen Zähler ein, der bei jedem Durchlauf der Logik um eins erhöht wird. Sobald dieser Zähler einen Wert von 4 erreicht, verlassen wir die Logik und setzen den `timerCounter` zurück. Bei 4 Durchläufen der Logik ist die effektive Wiederholrate bei 15 Bildern pro Sekunde. Und tiefer sollte unser Spiel nach Möglichkeit nicht rutschen.

```
maxSkip = 4;

// ...

curSkip = 0;
if (timerCounter > 0) {
    do {
        // Logik ...

        // Buchführung
        timerCounter--;
        curSkip++;
        if (curSkip >= maxSkip) {
            timerCounter = 0;
            break;
        }
    } while (timerCounter > 0);
    needsRefresh = TRUE;
}
```

Nachdem nun die Spiellogik mit einer konstanten Rate aufgerufen wird, können wir alle Animationen und Bewegungen genau festlegen.

12 Interaktion

Das Besondere an Computerspielen ist ihre Interaktivität. Bisher haben wir uns mit den Grundlagen der Darstellung von Grafiken und des Abspielens von Sound beschäftigt. Nun geht es darum, den Benutzer in das Geschehen einzubeziehen, und aus dem Betrachter einen Mitspieler zu machen.

Ein PC bietet dem Nutzer eine Vielzahl an Möglichkeiten, um mit ihm zu kommunizieren. Die Maus hat inzwischen die Tastatur in vielen Bereichen abgelöst und die Maus-/Tastaturkombination wird von den meisten PC-Spielern einer Lösung mit Gamepad oder Joystick vorgezogen. Doch gerade bei Rollenspielen im Konsolenstil oder bei Arkade-Titeln wird eine Eingabemöglichkeit mittels Joystick oder Joypad immer noch erwartet.

Im Laufe dieses Kapitels werden wir die Grundlagen beim Abfragen der gängigen Eingabegeräte erarbeiten.

Arten der Interaktion

In einem Spiel haben wie mehrere Möglichkeiten zur Interaktion. Die klassische Variante ist die direkte Kontrolle. Ich drücke den Knopf und die Figur auf dem Bildschirm springt. Bewege ich das Steuerkreuz nach rechts, bewegt sich auch die Figur nach rechts. Typische Beispiele für dieses Interaktionsmodell sind Action- und Hüpfspiele.

Etwas abstrakter ist die mittelbare Kontrolle. Ich gebe einen Zielpunkt vor, und die Spielfigur bewegt sich zu diesem Punkt. Anstatt die Spielfigur zu kontrollieren, wird sie nun befehligt. Typische Beispiele hierfür sind Abenteuerspiele wie »Monkey Island« oder (Echtzeit-)Strategiespiele wie »WarCraft3«. Insbesondere in Strategiespielen wird der Befehlscharakter deutlich. Die Einheiten scheinen ein Eigenleben zu haben und reagieren auf die Kommandos des Spielers auf ihre eigene Weise.

Bei der indirekten Kontrolle ist der Spieler nur noch ein Überwacher. Sein Einfluss beschränkt sich darauf die Spielfiguren zu unterstützen. So könnte er versuchen ihnen etwas beizubringen oder ihre Lebensumstände zu verbessern. Spiele wie »Die Sims« und »Black & White« fallen in diese Kategorie.

Zwischen den klassischen Beispielen dieser Modelle ist natürlich immer eine große Grauzone, in der Mischformen aller Art zu finden sind.

Direkte Kontrolle

Bei diesem Interaktionsmodell wird der Spieler direkt in das Spiel miteinbezogen. Jede Aktion seinerseits hat eine unmittelbare Reaktion des Spieles zur Folge. Dies gibt dem Spiel ein Actionelement und erlaubt es dem Spieler sich mit der Spielfigur zu identifizieren. Der Spieler wird sehr stark in das Spiel einbezogen.

Für ein Rollenspiel hat dies einige Vorteile. Der Spieler soll sich ja mit seinem Charakter identifizieren, also scheint die direkte Kontrolle erst einmal die beste Wahl zu sein. Allerdings werden durch den Actionanteil einige Spieler verschreckt. Klassische Rollenspiele wie »Terranigma« und »Zelda« haben die direkte Kontrolle benutzt. Und beiden wurde mehr als einmal vorgeworfen für ein Rollenspiel zu actionlastig zu sein.

Direkte Kontrolle lässt sich mit beinahe allen Eingabegeräten erreichen. Eine Bewegung der Maus ändert den Blickwinkel des Protagonisten in 1st Person Shootern, ein Tastendruck sorgt dafür, dass sich Joe der Dschungelheld über einen Abgrund schwingt, eine kleine Bewegung des Steuerknüppels lässt ein Raumschiff auf virtuelle Gegner herabjagen. Wer sich für dieses Eingabemodell entscheidet, hat die Qual der Wahl zwischen allen Eingabegeräten.

Mittelbare Kontrolle

Mittelbare Kontrolle ist beinahe immer ein mehrstufiger Prozess. Zuerst wird der Protagonist der Aktion ausgewählt (wenn denn mehrere zur Auswahl stehen), dann eine Aktion gewählt und schließlich ein Ort an dem diese Aktion stattfinden soll. Zum Teil wird das Auswählen auch halb-intelligent vom Programm übernommen. Klickt man auf eine leere Fläche, so bewegt sich der Charakter beispielsweise an diese neue Position. Klickt man auf einen Gegner, so greift man ihn an. Wählt man eine neutrale oder freundlich gesonnene Einheit, könnte ein Gespräch zu Stande kommen, während das Auswählen eines Gegenstandes dazu führt ihn einzustecken.

Dieses Interaktionsmodell eignet sich ebenfalls hervorragend für Rollenspiele. Zwar wird der Spieler nicht so stark einbezogen, die Flexibilität in der Aktionswahl macht dieses Manko allerdings wieder wett.

Die mittelbare Kontrolle schreit gerade nach einer Mausbedienung. Zwar sind andere Eingabemöglichkeiten denkbar, doch wird es in den meisten Fällen darauf hinaus laufen, mit Tastatur oder Joystick die Mausfunktionalität nachzuahmen.

Indirekte Kontrolle

Da der Spieler hier eine mehr oder weniger passive Rolle einnimmt, ist beinahe jedes Genre geeignet auf diese Weise gesteuert zu werden. Natürlich ist man dann auch in allen Genres zu der Rolle des indirekten Teilnehmers »verdammt«. In einem Rollenspielsetting könnte man beobachten, wie sich die Protagonisten verhalten, wenn sie mit den Taten des Spielers konfrontiert werden.

Alle Eingabemedien sind gleichermaßen für die indirekte Kontrolle geeignet. Die Kommandos können über Menüs, Mausgesten oder Joystickknöpfe gegeben werden – alles ist möglich und nur eine Frage des gewünschten Eingabestils.

Tastatur

Die Tastatur ist bei der Eingabe von Daten immer noch die unangefochtene Nummer 1. Mit knapp über 100 Tasten bietet sie eine Vielzahl an Möglichkeiten. Allerdings ist die Eingabe von Texten innerhalb eines Spieles heutzutage kaum noch nötig. Bestenfalls wenn es darum geht einen Namen für die Spielfigur zu vergeben oder sich in eine High-Score-Liste einzutragen wird die Tastatur noch im traditionellen Sinne verwendet.

Ansonsten wird die Tastatur in einem Spiel meist als eine Art Gamepad mit 102 Tasten betrachtet.

Bevor wir auf die Allegro Tastatur-Funktion zugreifen können, müssen wir diesen Teil der Bibliothek erst initialisieren. Ein schmerzloses

```
install_keyboard();
```

erledigt diesen Teil der Aufgabe. Nach diesem Aufruf können Sie nur noch die Allegro Funktionen zur Tastaturbehandlung benutzen. Wenn Sie versuchen nach `install_keyboard()` mit den normalen C/C++ Funktionen auf das Keyboard zuzugreifen, dann ist das Ergebnis undefiniert.

Ein Joypad mit 101 Tasten

Allegro stellt eine einfach zu nutzende Schnittstelle zur Verfügung, um den Zustand jeder einzelnen Taste abzufragen: das key-Array.

Wenn wir zu irgendeinem Zeitpunkt wissen wollen, ob derzeit die Leertaste gedrückt ist, dann reicht eine einfache Abfrage:

```
if (key[KEY_SPACE]) {
    // Reaktion auf Tastendruck
}
```

Auf diese Weise lässt sich jede Taste abfragen. Ein wichtiger Punkt ist, dass diese Funktion auf einer Low-Level-Ebene arbeitet. Die verwendeten Konstanten gehen also per Voreinstellung von einer US-amerikanischen Tastaturbelegung aus.

```
if (key[KEY_Z]) {
    // KEY_Z ist auf amerikanischen Keyboards
    // links neben dem "X".
}
```

Für jede Taste auf dem Keyboard gibt es eine entsprechende KEY_*-Konstante. Eine vollständige Liste finden Sie in der Tabelle »Tasten und Allegro-Konstanten« weiter unten.

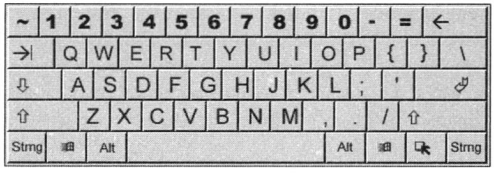


Abbildung 12.1: Übersicht über das US-Amerikanische Tastatur-Layout

Konstante	Taste / Bedeutung
KEY_ESC	Die Esc-Taste
KEY_F1 – KEY_F12	Die Funktionstasten
KEY_TILDE	Die Taste links von der 1 (Auf amerikanischen Tastaturen ist dies die Tilde »~«.)

Konstante	Taste / Bedeutung
KEY_0 – KEY_9	Zahlentasten
KEY_MINUS	Die Minustaste rechts neben der 0 (auf der deutschen Tastatur ist hier das »ß«.)
KEY_EQUALS	Das Gleichheitszeichen links neben der ⇦-Taste. (Auf der deutschen Tastatur ist hier das Hochkomma.)
KEY_BACKSPACE	Die ⇦-Taste
KEY_TAB	Die ⇥-Taste
KEY_A – KEY_Z	Die Buchstaben. Bitte beachten Sie die US-amerikanische Anordnung. Z und Y sind vertauscht.
KEY_OPENBRACE	Die sich öffnende geschweifte Klammer (bei deutschen Tastaturen das Ü)
KEY_CLOSEBRACE	Die sich schließende geschweifte Klammer (bei deutschen Tastaturen das +-Zeichen)
KEY_ENTER	Die ↵-Taste
KEY_CAPSLOCK	Die ⇩-Taste
KEY_LSHIFT	Die ⇧-Taste auf der linken Seite
KEY_COMMA	,
KEY_STOP	.
KEY_SLASH	Die Schrägstrich »/«-Taste (⇧ + 7). Auf deutschen Tastaturen ist hier der Unterstrich »_«.
KEY_RSHIFT	Die rechte ⇧-Taste
KEY_LCONTROL	Die linke Strg-Taste
KEY_LWIN	Die linke ⊞-Taste

Konstante	Taste / Bedeutung
KEY_LALT	Die linke [Alt]-Taste
KEY_SPACE	Die [Leer]-Taste
KEY_ALTGR	Die rechte [Alt]-Taste ([AltGr])
KEY_RWIN	Die rechte [⊞]-Taste
KEY_MENU	Die [▤]-Taste
KEY_RCONTROL	Die rechte [Strg]-Taste
KEY_PRTSCR, KEY_SCRLOCK, KEY_PAUSE	Die 3 Tasten rechts neben den Funktionstasten ([Druck], [Rollen] und [Pause])
KEY_INSERT, KEY_HOME, KEY_PGUP, KEY_DEL, KEY_END, KEY_PGDWN	Die 6 Tasten zwischen Haupttastenblock und Zifferntastenblock
KEY_UP, KEY_DOWN, KEY_RIGHT, KEY_LEFT	Die Tasten [→], [←], [↑] und [↓]
KEY_NUMLOCK	Die [Num]-Taste
KEY_SLASH_PAD	Das Divisionszeichen im Nummernblock
KEY_ASTERISK	Das Multiplikationszeichen im Nummernblock
KEY_MINUS_PAD	Das Minuszeichen im Nummernblock
KEY_PLUS_PAD	Das Pluszeichen im Nummernblock
KEY_ENTER_PAD	Das Enterzeichen im Nummenblock
KEY_0_PAD – KEY_9_PAD	Zahlen im Nummerneingabeblock
KEY_DEL_PAD	Die Löschtaste [Entf] im Nummerneingabeblock

Tabelle 12.1: Tasten und Allegro-Konstanten

Mit Hilfe des key Arrays kann man auf einfache Weise abfragen, ob eine oder mehrere Tasten gedrückt sind.

Es gibt bei der gleichzeitigen Abfrage mehrere Tasten ein Hardwareproblem: Die meisten der zur Zeit erhältlichen Tastaturen benutzen im wesentlichen die gleiche Abfrage für ihre Tastaturen. Und bei den meisten können ab einer bestimmten Anzahl von gedrückten Tasten weitere Tastendrücke nicht mehr erkannt werden. Aus diesem Grund ist es empfehlenswert, für Funktionen, die häufig in Kombination benutzt werden die ⇧-, Strg- und Alt-Tasten zu verwenden.

Diese können immer gemeinsam ausgelesen werden und sind aus diesem Grund »sicher.«

Drücken und Loslassen einer Taste

Das key Array gibt uns den aktuellen Zustand einer Taste zurück. In manchen Situationen ist es aber nur entscheidend zu wissen wann (und ob) eine Taste gedrückt wird, und wann sie wieder losgelassen wird. Ein typisches Beispiel hierfür sind Menüs, in denen man, um drei Einträge nach oben zu kommen, auch 3 mal die entsprechende Richtungstaste drücken muss.

Will man dieses Verhalten in einem Allegro Programm, so muss ein paar wenige Zeilen codieren.

```
const int KEY_UNCHANGED = 0;
const int KEY_PRESSED = 1;
const int KEY_RELEASED= 2;

int getKeyState(int keycode) {
    static int keyState[KEY_MAX];
    static int firstTime = 1;

    if (firstTime) {
        firstTime = 0;
        memset(keyState, 0, sizeof(int) * KEY_MAX);
    }
    if (key[keycode]) {
        if (keyState[keycode]) {
            return KEY_UNCHANGED;
        } else {
            keyState[keycode] = 1;
        }
```

```
    } else {
        if (keyState[keycode]) {
            keyState[keycode] = 0;
            return KEY_RELEASED;
        } else {
            return KEY_UNCHANGED;
        }
    }
}
```

Zuerst definieren wir 3 Konstanten für die möglichen Zustandsänderungen: KEY_UNCHANGED ist für den Fall, dass sich nichts geändert hat. Die Taste ist immer noch gedrückt oder immer noch im Ruhezustand. KEY_PRESSED ist für den Fall, dass die Taste gerade gedrückt wurde, KEY_RELEASED schließlich zeigt an, dass die Taste gerade losgelassen wurde.

Die getKeyState() Funktion benutzt ein zweites Array mit der gleichen Anzahl von Einträgen wie das key Array (KEY_MAX Elemente). Dieses Array ist static, das heißt die Werte innerhalb des Arrays bleiben zwischen den Aufrufen der Funktion erhalten. Es gibt nur ein Problem: Der Inhalt des Arrays ist nicht definiert. Je nachdem, wo das Array im Speicher liegt, können die unterschiedlichsten Werte enthalten sein.

Um nun sicher zu stellen, dass das Array mit gültigen Werte initialisiert wird, gibt es eine zweite statische Variable: firstTime.

Beim ersten Aufruf der Funktion hat firstTime den Wert 1. Die Bedingung der if()- Abfrage ist also wahr, und es passieren 2 Dinge:

- ✔ firstTime wird auf 0 gesetzt. Das bedeutet auch, dass die Bedingung der if()-Abfrage nie mehr wahr ergibt.
- ✔ Das Array wird mit Nullen gefüllt. Bis zu diesem Zeitpunkt war der Inhalt des Arrays nicht definiert.

Mit anderen Worten: Der zur if()-Abfrage gehörende Code wird nur einmal, beim ersten Aufruf der Funktion ausgeführt.

Ist nun die zu prüfende Taste gedrückt, wird überprüft, ob der entsprechende Eintrag im keyState Array gesetzt ist. Ist dies der Fall, dann hat sich am Zustand der Taste seit dem letzten Aufruf der Funktion nichts geändert. Doch wenn der Eintrag in keyState nicht gesetzt ist, dann bedeutet dies, dass die Taste beim letzten Aufruf der Funktion nicht gedrückt war. Also setzen wir das entsprechende Feld auf 1und geben KEY_PRESSED zurück.

Ist die zu prüfende Taste nicht gedrückt, dann brauchen wir nur zu prüfen, ob sie beim vorigen Aufruf gedrückt war. Ist der Eintrag in `keyState` gesetzt, dann wurde die Taste gerade losgelassen. In diesem Fall setzen wir `keyState[keycode]` auf 0 und geben `KEY_RELEASED` zurück.

Das `keyState` Array wird also dazu benutzt den Zustand der Taste zwischenzuspeichern. Durch den Vergleich des letzten Zustands mit dem jetzigen können wir die Änderung am Status feststellen.

Der Tastaturpuffer

Nun ist es sicherlich innerhalb des Spieles recht praktisch die Tasten einzeln abzufragen, doch wenn man einfach nur eine Texteingabe von der Tastatur entgegennehmen möchte, wäre es sehr umständlich dies über das `key` Array zu erledigen. Zum Glück bietet uns Allegro auch für diese Aufgabe eine Reihe nützlicher Funktionen an.

Die drei grundlegenden Funktionen für diesen Fall sind:

```
int keypressed();
int readkey();
void clearkeybuf();
```

Mit diesen 3 Funktionen lassen sich 80% der Fälle abdecken. Die `keypressed()`-Funktion gibt einen von 0 verschiedenen Wert zurück, falls seit ihrem letzten Aufruf Tasten gedrückt wurden. Diese Tasten werden im Tastaturpuffer gespeichert. Am einfachsten stellt man sich diesen Puffer als ein Array vor, in das die Tasten in der Reihenfolge eingetragen werden, in der sie gedrückt wurden.

Ein Aufruf von `readkey()` gibt die Taste zurück, die am längsten im Puffer ist. Dadurch ist sicher gestellt, dass die Tasten auch in der Reihenfolge zurückgegeben werden, in der sie gedrückt wurden.

Der Rückgabewert von `readkey()` ist erklärungsbedürftig. Hier haben die Allegro-Entwickler etwas getrickst und geben eigentlich zwei Werte zurück, die nur zu einem `int` »gepackt« wurden. Die unteren 8 Bit des Rückgabewertes enthalten den ASCII-Wert des nächsten Buchstabens im Puffer, also das eigentliche Zeichen (`char`). Die nächsten 8 Bit enthalten den Scancode der Taste. Dieser Scancode entspricht den `KEY_*` -Konstanten, über die das `key` Array abgefragt wird. Der Scancode bleibt immer gleich, egal ob nun ⇧, Strg oder ⇧ gedrückt wird. Der Buchstabe in den unteren 8 Bit jedoch ändert sich aufgrund dieser Tasten. Wird zum Beispiel die Taste A gedrückt, so bleibt der Scancode `KEY_A`, egal ob nun

eine der Umschalttasten gedrückt wird oder nicht. Das Zeichen ändert sich jedoch wie Sie in der Tabelle »Einfluss der Umschalttasten« sehen können.

Taste	Zeichen / Wert
A	a (97)
⇧ + A	A (65)
Alt + A	0
Strg + A	1

Tabelle 12.2: Einfluss der Umschalttasten

Die ⇧-Taste ändert das Zeichen wie erwartet (Groß-/Kleinschreibung, alternatives Zeichen). Bei der Alt-Taste ist das Zeichen 0 und nur der Scancodeteil des Rückgabewertes ist gesetzt. Bei der Strg-Taste werden die unteren 8 Bit auf die Position des Zeichens im Alphabet gesetzt. Also A = 1, B = 2, C = 3, ... Z = 26.

Um den Rückgabewert zu splitten, muss man etwas in die binäre Trickkiste greifen. Um die unteren 8 Bits zu bekommen, muss man einfach alle anderen Bits auf 0 setzen. Dies erreicht man durch ein binäres UND mit `0xff`. Ein UND setzt alle Bits auf 1, die in beiden Operanden 1 sind, und setzt alle anderen Bits auf 0. Für die nächsten 8 Bit verschiebt man alle Bits um 8 Stellen nach links und verknüpft5 dann wieder mit 0xff.

```
int code = readkey();
int charCode = code & 0xff;
int scanCode = (code >> 8) & 0xff ;
```

Benutzt man `readkey()`, wird das gelesene Zeichen aus dem Tastaturpuffer entfernt. Manchmal ist es jedoch hilfreich, alle bisher im Puffer stehenden Eingaben einfach zu ignorieren. Zu diesem Zweck gibt es die `clearkeybuf()` Funktion. Sie setzt den Eingabepuffer zurück und stellt sicher, dass nur neue Eingaben berücksichtigt werden.

Damit haben wir die wichtigsten Funktionalitäten behandelt. Nun wird es Zeit für ein (kleines) Beispielprogramm, in dem all dies in die Praxis umgesetzt wird.

Tastatur-Beispielprogramm

In dem Beispiel dreht sich alles nur um die Eingabefunktionen. Schmückendes Beiwerk wie Timer-Funktionen wurden aus Übersichtsgründen entfernt.

Das Beispiel ist recht simpel: Die Cursortasten werden abgefragt, und je nach, dem welche Cursortaste gedrückt ist, wird ein Joystick in verschiedenen Positionen gezeigt.

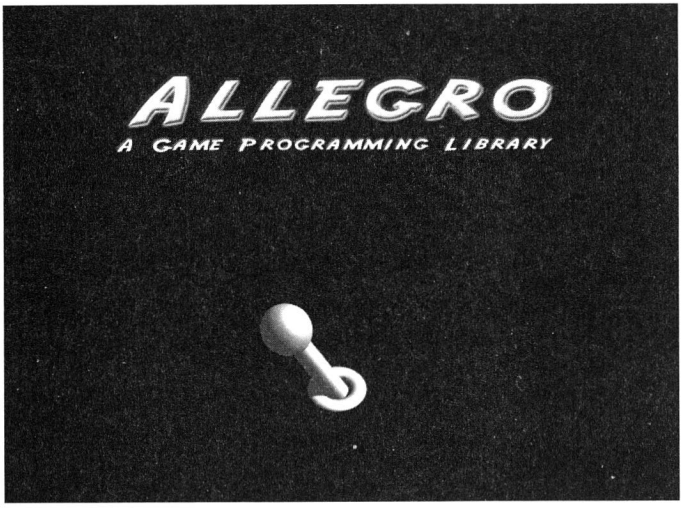


Abbildung 12.2: Screenshot vom Beispielprogramm

Durch Druck der F2-Taste wird vom normalen Modus (Tastenzustand) zur Abfrage der Zustandsänderung umgeschaltet. Ein Druck auf F1 setzt den Modus wieder zurück. Durch Drücken der Esc-Taste wird das Programm beendet.

```
#include <allegro.h>
#include <string.h>

const int COUNT_JOY_IMAGES = 5;

const int KEY_UNCHANGED = 0;
```

```
const int KEY_PRESSED = 1;
const int KEY_RELEASED= 2;

const int MODE_STATE = 0;
const int MODE_STATE_CHANGE = 1;

BITMAP *joystickBitmaps[COUNT_JOY_IMAGES];

int set_graphic_mode() {
    int depths[] = { 16, 15, 16};
    int modes[]  = {GFX_AUTODETECT_FULLSCREEN,
                    GFX_AUTODETECT_FULLSCREEN,
                    GFX_AUTODETECT_WINDOWED };

    for (int a=0; a < 3; a++) {
        set_color_depth(depths[a]);
        if (set_gfx_mode(modes[a], 640, 480, 0, 0) >= 0) {
            return 1;
        }
    }
    return 0;
}

int getKeyState(int keycode) {
    static int keyState[KEY_MAX];
    static int firstTime = 1;

    if (firstTime) {
        firstTime = 0;
        memset(keyState, 0, sizeof(int) * KEY_MAX);
    }
    if (key[keycode]) {
        if (keyState[keycode]) {
            return KEY_UNCHANGED;
        } else {
            keyState[keycode] = 1;
        }
    } else {
        if (keyState[keycode]) {
            keyState[keycode] = 0;
            return KEY_RELEASED;
        } else {
            return KEY_UNCHANGED;
        }
    }
```

```
}

int main(int argc, char **argv) {

    // ------------ Main ----------------------
    allegro_init();

    if (!set_graphic_mode()) {
        allegro_message("Unable to set graphic mode!");
        exit(0);
    }
    install_keyboard();

    BITMAP *doublebuffer = create_bitmap(SCREEN_W, SCREEN_H);
    BITMAP *logo = load_bitmap("allegro.tga", NULL);

    char   *filenames[] = {
              "joy_norm.bmp",
              "joy_left.bmp",
              "joy_right.bmp",
              "joy_up.bmp",
              "joy_down.bmp"};
    for (int a = 0; a < COUNT_JOY_IMAGES; a++) {
        joystickBitmaps[a] = load_bitmap(filenames[a], NULL);
    }

    // ------------ Main ----------------------
    int curImage = 0;
    int mode     = MODE_STATE;

    while (!key[KEY_ESC]) {

        if (key[KEY_F1]) {
            mode = MODE_STATE;
        } else if (key[KEY_F2]) {
            mode = MODE_STATE_CHANGE;
        }

        switch (mode) {
            case MODE_STATE:
                if (key[KEY_UP]) {
                    curImage = 3;
                } else if (key[KEY_DOWN]) {
                    curImage = 4;
                } else if (key[KEY_LEFT]) {
```

```
                    curImage = 1;
                } else if (key[KEY_RIGHT]) {
                    curImage = 2;
                } else {
                    curImage = 0;
                }
                break;


            case MODE_STATE_CHANGE:
                if (getKeyState(KEY_UP) == KEY_PRESSED) {
                    curImage = 3;
                } else if (getKeyState(KEY_DOWN) == KEY_PRESSED) {
                    curImage = 4;
                } else if (getKeyState(KEY_LEFT) == KEY_PRESSED) {
                    curImage = 1;
                } else if (getKeyState(KEY_RIGHT) == KEY_PRESSED) {
                    curImage = 2;
                } else {
                    curImage = 0;
                }
                break;
        }

        // ------------ Display ----------------------
        acquire_bitmap(doublebuffer);
        clear(doublebuffer);
        blit( logo, doublebuffer, 0, 0, (SCREEN_W - logo->w)/2, 55,
logo->w, logo->h);
        blit( joystickBitmaps[curImage], doublebuffer, 0, 0,
(SCREEN_W - joystickBitmaps[curImage]->w) /2, 245,
joystickBitmaps[curImage]->w, joystickBitmaps[curImage]->h);
        release_bitmap(doublebuffer);
        vsync();
        blit(doublebuffer, screen, 0, 0, 0, 0, doublebuffer->w,
doublebuffer->h);
    }

    destroy_bitmap(doublebuffer);
    for (int a= 0; a < COUNT_JOY_IMAGES; a++) {
        destroy_bitmap(joystickBitmaps[a]);
    }

} END_OF_MAIN();
```

Nach dem Initialisieren von Allegro setzen wir den Grafikmodus. Wie bisher versuchen wir erst einen Vollbildmodus in 16- oder 15-Bit-Farbdichte zu setzen. Haben wir damit keinen Erfolg setzen wir einen 16-Bit-Fenstermodus.

Nach der Initialisierung der Keyboardroutinen werden die Grafiken geladen. Nachdem noch die Variablen für das aktuelle Bild des Joysticks und den Abfragemodus des Keyboards gesetzt sind, kann es mit der Hauptschleife losgehen.

Je nach gesetztem Modus wird nun das anzuzeigende Bild entweder direkt über das `key` Array oder durch Aufruf der `getKeyState()`Funktion ermittelt.

Schließlich wird alles noch angezeigt und die Schleife beginnt von neuem.

Drücken von Esc beendet das Programm. Bevor das Programm verlassen wird geben wir noch die geladenen Bilder durch einen Aufruf von `destroy_bitmap()` wieder frei.

Und das war es.

Spielen Sie noch etwas mit diesem Beispiel herum. Ändern Sie den Quellcode so ab, dass Sie das Allegro Logo mit den Cursortasten bewegen können. Sie können auch den Timing Code aus dem letzten Kapitel in dieses Programm integrieren.

Die Maus

Normalerweise können Sie davon ausgehen, dass jeder PC mit einer Maus ausgestattet ist. Und wenn das Spielgenre es zulässt, dann sollte man die Maus auch unterstützen. Gerade in Rollenspielen kann die Maus sehr nützlich sein. Ob es nun darum geht einen Gegner auszuwählen oder einfach nur festzulegen, wo die Spielfigur hinlaufen soll – mit der Maus ist es nur einen Klick entfernt.

Das Einbinden der Maus funktioniert im Großen und Ganzen so wie bei der Tastatur: Zuerst wird eine Routine aufgerufen, um das Maus-Subsystem zu initialisieren. Ab diesem Zeitpunkt kann man alle Maus-Funktionen aufrufen, die Position des Mauszeigers abrufen etc.

Funktion	Beschreibung
install_mouse()	Gibt die Anzahl der Knöpfe zurück, die sich auf der Maus befinden. Ist keine Maus vorhanden, ist der Rückgabewert –1.
show_mouse(BITMAP *bmp)	Zeigt den Mauszeiger an. Die übergebene `BITMAP` sollte normalerweise der `screen` Zeiger sein. Übergibt man `NULL`, verschwindet der Mauszeiger. Um diese Funktion nutzen zu können, muss der Timer initialisiert worden sein.
mouse_x	Die aktuelle X-Position des Mauszeigers
mouse_y	Die aktuelle y-Position des Mauszeigers
mouse_b	Der aktuelle Zustand der Mausknöpfe

Tabelle 12.3: Die wichtigsten Mausfunktionen und Variablen

Nachdem die Maus initialisiert wurde, kann man sofort die Position des Mauszeigers abfragen – auch wenn der Mauszeiger nicht auf dem Schirm zu sehen ist.

Mit dem folgenden Code können Sie die Mausposition anhand der Schnittposition zweier Linien nachvollziehen.

```
install_mouse();
while (!key[KEY_ESC]) {
    // ------------ Display ----------------------
    acquire_bitmap(doublebuffer);
    clear(doublebuffer);
    line(doublebuffer, mouse_x, 0, mouse_x, SCREEN_H,
         makecol(255, 255, 255));
    line(doublebuffer, 0, mouse_y, SCREEN_W, mouse_y,
         makecol(255, 255, 255));
    release_bitmap(doublebuffer);
    vsync();
    blit(doublebuffer, screen, 0, 0, 0, 0,
         doublebuffer->w, doublebuffer->h);
}
```

Dieses Beispiel zeigt neben der Abfrage der Mausposition auch die Verwendung von Allegros `line()` Funktion. Diese Funktion malt eine Linie vom Startpunkt zum (Überraschung!) Endpunkt. Der letzte Parameter ist die Farbe, in der die Linie gemalt wird. Der Aufruf von `makecol()` ermittelt anhand der übergebenen RGB-Werte den korrekten Farbwert.

Neben Linien gibt es noch eine Vielzahl anderer grafischer Grundfunktionen, die Sie in Ihrem Programm benutzen können. Ich werde auf diese Funktionen nicht gesondert eingehen, sondern sie erläutern, sobald sie zum ersten Mal benutzt werden.

Eine weitere neue Funktion ist `vsnyc()`. Diese Funktion wartet bis der Kathodenstrahl des Bildschirms an der unteren rechten Ecke ist, und verhindert dadurch, dass die Linien zerrissen werden.

Der Kathodenstrahl bewegt sich von links nach rechts und von oben nach unten über den Bildschirm. Wenn nun das Bild beim Anzeigen vom Kathodenstrahl überholt wird, dann zeigt der obere Bereich des Schirms das neue Bild an, der untere noch das alte. Dieses Zerreißen des Bildes ist vor allem bei kontrastreichen vertikalen Linien, die sich schnell bewegen zu sehen.

Anstatt der Linien könnte man auch ein Bild anzeigen – womit wir beim Thema Mauszeiger wären.

Mauszeiger ändern

Ein Mauszeiger ist nichts anderes als eine Grafik, die an der aktuellen Mausposition angezeigt wird.

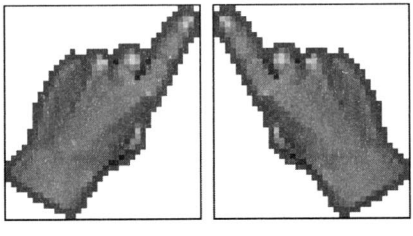


Abbildung 12.3: Vergrößerung von 2 Mauszeigern

Stellen wir uns vor, wir würden den linken Mauszeiger in einem Programm benutzen. Wenn wir dieses Bild an der Mausposition anzeigen, haben wir ein kleineres Problem: Der Finger, der auf die Position zeigt, ist etwa 36 Pixel weit von der Mausposition entfernt. Wenn man mit diesem Zeiger auf ein kleines Element zeigt und dann klickt, ist die Chance recht hoch, dass die eigentliche Mausposition außerhalb des Elements liegt.

Beim rechten Mauszeiger ist dieses Problem nicht gegeben, da sich Anzeigeposition und Position des Fingers an ungefähr der gleichen Stelle befinden. Doch was, wenn einem nun die linke Grafik besser gefällt? Kein Problem, in diesem Fall wird einfach die Grafik leicht versetzt zur eigentlichen Mausposition angezeigt, damit die Stelle auf die der Mauszeiger zeigt, und die numerische Position der Maus wieder übereinstimmen.

Man definiert also einen Offset für die Anzeige des Cursors. Dieser Offset ist leicht zu finden. Man sucht einfach die Position innerhalb des Cursors, mit dem gezeigt werden soll. Im Fall des linken Cursors wäre dies die Position (34,2). Der Offset ergibt sich nun durch das Multiplizieren mit –1, also (-34, -2).

Um also diesen Cursor korrekt anzuzeigen, brauchen wir folgenden Aufruf:

```
const int mouse_ofs_x = -34;
const int mouse_ofs_y = -2;

draw_sprite(doublebuffer, pointer,
    mouse_x + mouse_ofs_x,
    mouse_y+ mouse_ofs_y);
```

Die `draw_sprite()` Funktion arbeitet im Prinzip so wie `blit()`. Es gibt nur zwei kleine Unterschiede:

- ✔ `draw_sprite()` zeigt immer die komplette Bitmap an.
- ✔ Alle Pixel, die in hellem Pink (RGB: 255,0,255) gemalt sind, werden nicht dargestellt. Diese Pixel sind also transparent.

Würden wir `blit()` anstatt von `draw_sprite()` benutzen, dann würde der gesamte rechteckige Bereich der Bitmap angezeigt werden, anstatt nur die eigentliche Hand.

Auf Sprites werden wir in Kürze noch näher eingehen.

Es gibt in Allegro zwei Funktionen, die einem genau diese Arbeit abnehmen sollen: `set_mouse_sprite()` zum Ändern des Cursors und `set_mouse_focus()`, um den Hot-Spot innerhalb des Sprites festzulegen. Leider funktionieren diese beiden Funktionen zur Zeit des Schreibens nicht auf allen Plattformen zuverlässig. Ich gehe davon aus, dass diese Probleme in der Zwischenzeit behoben sind, verwende aber trotzdem in allen Kapiteln die oben beschriebene Methode.

Ein interessantes Beispielprogramm für eine Mausabfrage zu finden war keine leichte Aufgabe. Einfach dem Mauszeiger beim Bewegen über den Schirm zuzusehen ist einfach nicht sonderlich spannend.

Aus diesem Grund wurde das Programm etwas aufgepeppt. Im Hintergrund ist eine Weltkarte zu sehen. Der Bereich in dem sich der Mauszeiger befindet, wird vergrößert dargestellt.

Man bewegt also eine Lupe über die Weltkarte. Und schon wird das Bewegen des Mauszeigers etwas interessanter.

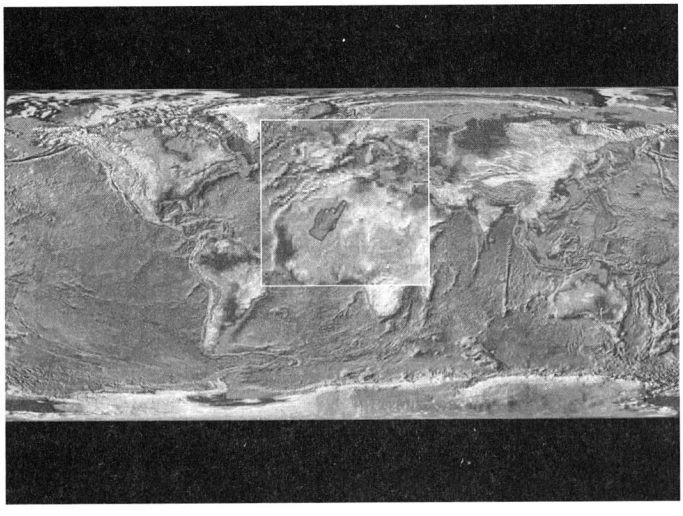


Abbildung 12.4: Screenshot des Mausabfrage-Beispieles

Die Weltkarte ist in einem Breitbildformat, am oberen und unteren Ende ist also noch etwas Platz. Um zu verhindern, dass sich die Maus aus dem Bereich der Karte herausbewegt, benutzen wir die set_mouse_range()-Funktion. Ihr werden vier Parameter übergeben, welche die gegenüberliegenden Eckpunkte eines Rechtecks bilden. Der Mauszeiger kann sich nur innerhalb dieses Bereichs bewegen.

Um den Bereich unterhalb des Cursors zu vergrößern, benutzen wir die Funktion

```
stretch_blit(BITMAP*source, BITMAP*dest,
    int source_x, int source_y,
    int source_width, int source_height,
    int dest_x, int dest_y,
    int dest_width, int dest_height)
```

Die Funktion bildet den Bereich der source Bitmap, der durch (source_x, source_y) und (source_width, source_height) definiert wird auf der dest Bitmap an der Stelle (dest_x, dest_y) mit der Größe (dest_width, dest_height) ab.

Oder anders gesagt: Das angegebene Rechteck der source Bitmap wird so skaliert, dass es in das angegebene Rechteck der dest Bitmap passt.

```
while (!key[KEY_ESC]) {
    // ------------ Display ----------------------
    clear(doublebuffer);
    blit(bg, doublebuffer, 0, 0, 0, 0, bg->w, bg->h);
    if (mouse_y >= 80 && mouse_y <= SCREEN_H-80) {
        stretch_blit(bg, doublebuffer,
            mouse_x-40, mouse_y-40, // source x/y
            80, 80,                 // source w/h
            mouse_x-80, mouse_y-80, // dest x/y
            160, 160);              // dest w/h
        rect(doublebuffer,
             mouse_x -80, mouse_y-80,
             mouse_x + 80, mouse_y +80,
             makecol(255,255,255));
    }
    draw_sprite(doublebuffer, pointer,
        mouse_x + mouse_ofs_x, mouse_y+ mouse_ofs_y);
    vsync();
    blit(doublebuffer, screen, 0, 0, 0, 0, doublebuffer->w,
doublebuffer->h);
}
```

Um den vergrößerten Bereich von der restlichen Karte abzusetzen, zeichnen wir noch ein weißes Rechteck um diesen Bereich.

Den vollständigen Sourcecode dieses Beispieles finden sie auf der beigelegten CD.

Zum Kompilieren benutzen Sie bitte folgende Befehlszeile:

```
g++ -o prog12_3.exe prog12_3.cpp -lalleg
```

Mausknöpfe und Räder

Wie man die Mausposition abfragen kann, ist also nun kein Geheimnis mehr. Doch was ist mit den Mausknöpfen? Und eventuell einem Mausrad? Auch dies ist ein Kinderspiel. Allegro gibt beim Aufruf von `install_mouse()` die Anzahl der verfügbaren Mausknöpfe zurück. Der Zustand dieser Knöpfe wird in der Variablen `mouse_b` gehalten. Für jeden Mausknopf ist ein Bit reserviert.

```
if (mouse_b & 0x01) {
    // Linker Mausknopf
}
if (mouse_b & 0x02) {
    // Rechter Mausknopf
}
if (mouse_b & 0x04) {
    // Mittlerer (3ter) Mausknopf
}
```

Die Bitmaske für den ersten Mausknopf (in der Regel der linke) ist 0x01. Der zweite Knopf hat eine Maske von 0x02, dann 0x04, 0x08, 0x10 und so weiter.

Durch diese Regelung kann man mit einer einzelnen Abfrage auch beliebige Kombinationen von Knöpfen abfragen:

```
if (mouse_b & 0x05) {
    // linker und mittlerer Mausknopf
    // (0x01 + 0x04)
}
```

Das Mausrad hingegen arbeitet eher wie eine weitere Achse. Die aktuelle Position der »Scrollachse« wird in `mouse_z` gespeichert. Im Gegensatz zu den normalen Achsen `mouse_x` und `mouse_y` kann `mouse_z` auch negative

Werte enthalten. Normalerweise ist der tatsächliche Wert von mouse_z weniger interessant als die Veränderung des Wertes, da diese eine Bewegung des Rades anzeigt. Das bisherige Beispiel lässt sich sehr leicht umwandeln, um mit Hilfe des Scrollrades eine verstellbare Zoomfunktion zu erreichen.

Der erste Schritt besteht darin, die bisher hartcodierten Werte für die Größe des Zielbereiches durch Variablen zu ersetzen.

```
int dest_w = 160;
int dest_h = 160;
```

Dann muss noch der Aufruf der stretch_blit() und rect() Funktionen angepasst werden, damit diese Variablen auch benutzt werden.

Wenn sich nun das Mausrad bewegt, muss sich nun noch die Größe des Zielbereichs entsprechend verändern. Aus diesem Grund wird die Differenz des bisherigen Wertes vom aktuellen Wert zu der Größe dazu gezählt.

```
int delta_zoom = mouse_z - cur_z;
dest_w += delta_zoom;
dest_h += delta_zoom;
cur_z   = mouse_z;
dest_w  = MID(10, dest_w, 400);
dest_h  = MID(10, dest_h, 400);
```

Am Ende stellen wir noch sicher, dass die Werte für dest_w und dest_h innerhalb eines sinnvollen Bereiches liegen. MID funktioniert auf folgende Weise, ist aber als Makro definiert:

```
int MID(int min, int cur, int max) {
    if (cur < min) {
        return min;
    }
    if (cur > max) {
        return max;
    }
    return cur;
}
```

Der Rückgabewert von MID() ist also immer innerhalb vom min und max. In unserem Fall begrenzen wir die Zielgröße auf ein Minimum von 10 Pixeln und ein Maximum von 400 Pixeln.

Eine weitere kleine Änderung ist die Abbruchbedingung der Hauptschleife: Sobald eine Maustaste betätigt wird, wird in diesem Beispiel das Programm beendet.

Den vollständigen Quellcode für dieses Programm finden Sie als prog12_4.cpp auf der beliegenden CD.

Zum Kompilieren benutzen Sie bitte folgende Befehlszeile:

```
g++ -o prog12_4.exe prog12_4.cpp -lalleg
```

Der Joystick

Das Eingabegerät, das bei den Konsolen unangefochten an erster Stelle steht, führt am PC ein Schattendasein. Joysticks und Joypads haben sich nur in Nischen durchsetzen können. Der Grund dafür ist meiner Meinung nach einfach, dass es keinen einheitlichen Standard gibt.

Zu DOS-Zeiten war es noch einfach. Ein Joystick hatte in der Regel zwei Tasten und einen analogen Knüppel. Dann kamen die ersten 4-Tasten-Pads. Dank DirectX gibt es heute keine Beschränkungen bezüglich der Anzahl der Achsen und Buttons mehr.

Aus diesem Grund kann sich ein Entwickler nicht darauf verlassen, welche Features ein Joystick nun hat. Die Folge war, dass sich die meisten Spielehersteller auf Maus und Tastatur konzentriert haben. Dadurch kamen weniger Spiele auf den Markt, die sich mittels Joystick steuern ließen. Aus diesem Grund kauften sich weniger Leute Joysticks. Die sinkenden Verkaufszahlen der Joysticks sorgten dafür, dass die Hersteller noch weniger Wert auf Joystickunterstützung legten und diese Spirale dreht sich immer weiter.

Derzeit werden Joysticks meist nur noch für Flugsimulationen eingesetzt und Joypads für Sport und Hüpfspiele. In beiden Fällen werden sich wohl nur echte Fans ein Pad kaufen.

Dies ist ein echter Jammer, denn Joypads haben einen großen Vorteil: Man kann jede Menge von ihnen an einen PC anschließen (USB sei Dank) und dann mit mehreren Personen an einem PC spielen.

Und selbst das normale Gamepad mit nur 4 Tasten gibt genug Möglichkeiten für die meisten Spiele.

Aus diesem Grund möchte ich Sie ermuntern in ihren Spielen Joysticks und Gamepads zu unterstützen. Es ist kein Problem, die Unterstützung in Ihre Programme zu integrieren und alle Besitzer eines Pads werden es Ihnen danken.

Joystick-Support aktivieren

Wie immer ist das Aktivieren der Unterstützung durch einen einzelnen Aufruf erledigt:

```
install_joystick(JOY_TYPE_AUTODETECT)
```

Aktiviert den Support für Joysticks. Nach diesem Aufruf haben Sie vollen Zugriff auf alle Joystick Funktionen. Die Anzahl der angeschlossenen Joysticks steht in

```
num_joysticks
```

Die einzelnen Informationen (welche Taste gedrückt ist und in welche Richtung gelenkt wird) können aus dem `joy[]` Array ausgelesen werden. Allerdings werden diese Informationen nicht automatisch auf den neuesten Stand gebracht. Sie müssen in Ihrer Hauptschleife die Funktion

```
poll_joystick()
```

aufrufen, damit die Zustandsinformationen aktualisiert werden.

Das Joystick-Array

Im `joy`-Array sind alle angeschlossenen Joysticks vertreten. Jeder Eintrag in diesem Array hat den folgenden Aufbau:

```
typedef struct JOYSTICK_INFO {
    // Informationen über den Joystick
    int flags;
    // Wieviele Steuerknüppel
    int num_sticks;
    // Wieviele Knöpfe
    int num_buttons;

    // Zustand aller vorhanden Knüppel
    JOYSTICK_STICK_INFO stick[n];
    // Zustand aller vorhanden Knöpfe
    JOYSTICK_BUTTON_INFO button[n];
} JOYSTICK_INFO;
```

Jetzt folgt eine Menge notwendiger Informationen. Ein Joystick kann heutzutage eine äußerst komplexe Angelegenheit sein. Ich würde deswegen empfehlen, dass Sie sich erst einen groben Überblick verschaf-

fen. Lesen Sie die folgenden Informationen durch, und nehmen Sie sie einfach nur zur Kenntnis. Im Anschluss an diese Übersicht folgen praktische Beispiele, welche die Zusammenhänge und die Nutzung des Joysticks erklären.

Der erste Eintrag, `flags`, enthält einige Zusatzinformationen über den Joystick. Es ist ein Bitfeld, in dem eine Kombination der folgenden Bits auftreten kann:

Name	Beschreibung
`JOYFLAG_DIGITAL`	Gerät ist im digitalen Eingabemodus.
`JOYFLAG_ANALOGUE`	Gerät ist im analogen Eingabemodus.
`JOYFLAG_CALIB_DIGITAL`	Kann digitale Eingaben nach der Kalibrierung erzeugen.
`JOYFLAG_CALIB_ANALOGUE`	Gerät kann nach der Kalibrierung analoge Daten zurückliefern.
`JOYFLAG_CALIBRATE`	Gerät muss kalibriert werden.
`JOYFLAG_SIGNED`	Die Achsendaten sind vorzeichenbehaftet und im Bereich –128 bis 127
`JOYFLAG_UNSIGNED`	Die Achsendaten sind nicht vorzeichenbehaftet und im Bereich 0 bis 255.

Tabelle 12.4: Liste der Joystick Flags

Die meisten analogen Joysticks kalibrieren sich entweder beim Starten des Rechners automatisch oder können über ein Betriebssystem-Utility kalibriert werden. Aus diesem Grund ist es auf Linux und Windows-Systemen meist nicht nötig, eine Kalibrierung durchzuführen. Insbesondere wenn man sich auf die digitale Eingabe konzentriert.

Die Anzahl der vorhandenen Steuerknüppel wird in `num_sticks` gespeichert. Ein normaler Joystick hat nur einen Knüppel, ein »Flightstick« könnte mehrere haben. Zum Beispiel jeweils einen für Lage und Schubkontrolle. Egal wie viele Knüppel nun ein Joystick hat, sie können davon ausgehen, dass der erste im Array der primäre Stick ist. Jeder Knüppel

hat nun eine bestimmte Anzahl an Achsen. Ein »normaler« Joystick hat 2 Achsen (horizontal und vertikal), eine Schubkontrolle an einem Flightstick hat meist nur eine.

Name	Beschreibung
`Flags`	Informationen zur Art des Sticks
`num_axis`	Anzahl der unterstützten Achsen
`axis[]`	Achsenpositions-Informationen. Die Größe dieses Array entnehmen Sie `num_axis`.
`name`	Zeichenkette mit dem Namen des Sticks

Tabelle 12.5: Die JOYSTICK_STICK_INFO-Struktur

Die Position lässt sich sowohl digital als auch analog abfragen – unabhängig vom eigentlichen Joysticktyp. Natürlich ist es meist nicht sehr sinnvoll, eine digitale Achse analog abzufragen – aber möglich wäre es. Die JOYSTICK_AXIS_INFO Struktur enthält alle notwendigen Informationen über die Achse:

Name	Beschreibung
`pos`	Analoger Wert dieser Achse
`d1, d2`	Digitale Information. `d1` entspricht dem minimalen Wert (z.B. Links), `d2` dem maximalen (z.B. Rechts).
`Name`	Zeichenkette mit dem Namen dieser Achse

Tabelle 12.6: Die JOYSTICK_AXIS_INFO-Struktur

Genauso wie die Anzahl der Knüppel ist auch die Anzahl der Buttons auf einem Joystick nach oben offen. Das `button` Array hat für jeden der vorhandenen Joystickknöpfe einen Eintrag. Nur die ersten `num_buttons` Einträge enthalten auch sinnvolle Daten.

Name	Bedeutung
b	Logischer An- / Aus- Wert. Wenn der Button nicht gedrückt ist, dann ist `b == 0`.
name	Zeichenkette mit dem Namen des Buttons

Tabelle 12.7: Die JOYSTICK_BUTTON_INFO-Struktur

Nach all diesen Strukturen wird es nun Zeit, dass wir endlich mal ans Programmieren gehen.

Abfrage der Buttons

Um einen bestimmten Button abfragen zu können, müssen wir zwei Dinge wissen:

- ✔ Der Joystick, auf dem er sich befindet
- ✔ Der Index im `button`-Array dieses Joysticks

Sobald diese Informationen vorhanden sind, ist das Abfragen des Joysticks ein Kinderspiel.

```
inline int checkJoyButton(int joystick, int index) {
    return joy[joystick].button[index].b;
}
```

Mit dieser kleinen Helferfunktion kann man die Buttons nun sehr einfach abfragen. Das `inline` Keyword bewirkt, dass diese Funktion schnell aufgerufen wird.

Genauer gesagt bewirkt `inline`, dass die Funktion überhaupt nicht aufgerufen wird. Stattdessen wird sie beim Erstellen des Programms an den Positionen direkt eingefügt, an denen der Aufruf steht. `Inline`-Funktionen sind die C++ Entsprechung der C Makros.

Abfrage der Joystick-Richtungen

Egal wie viele Knüppel ein Joystick hat, der Hauptjoystick ist immer an Position 0 im `sticks` Array. Und die Reihenfolge der Achsen ist im Falle des ersten Knüppels immer x,y.

Wenn der Joystick nach links gedrückt wird, dann ist der untere digitale Wert (d1) der ersten Achse gesetzt. Drückt man ihn nach rechts, dann wird der obere Wert (d2) gesetzt.

Beim Drücken nach oben oder unten verhält es sich entsprechend, nur das diesmal die Werte der zweiten Achse gesetzt werden.

```
// Konstanten für die einzelnen Richtungen
const int JOY_LEFT  = 1;
const int JOY_RIGHT = 2;
const int JOY_UP    = 3;
const int JOY_DOWN  = 4;

int checkJoyDir(int joystick, int dir) {
    switch (dir) {
        case JOY_LEFT:
            return joy[joystick].stick[0].axis[0].d1;
            break;
        case JOY_RIGHT:
            return joy[joystick].stick[0].axis[0].d2;
            break;
        case JOY_UP:
            return joy[joystick].stick[0].axis[1].d1;
            break;
        case JOY_DOWN:
            return joy[joystick].stick[0].axis[1].d2;
            break;
    }
    return 0;
}
```

Mit diesen beiden Funktionen können Sie nun den Joystick nach Belieben abfragen.

Testprogramm

Jetzt fügen wir alles zusammen und erstellen ein Testprogramm, das uns anzeigt, in welche Richtung der Joystick gedrückt wird und welche Buttons derzeit aktiviert sind.

Für jede der neun möglichen Positionen des Joysticks (acht Richtungen + Ruhestellung) wurden Grafiken gerendert. Die Grafik für die Buttons wird jedoch einfach wiederverwendet. Zwar ist es dann perspektivisch nicht vollkommen korrekt, doch dies fällt normalerweise nicht auf.

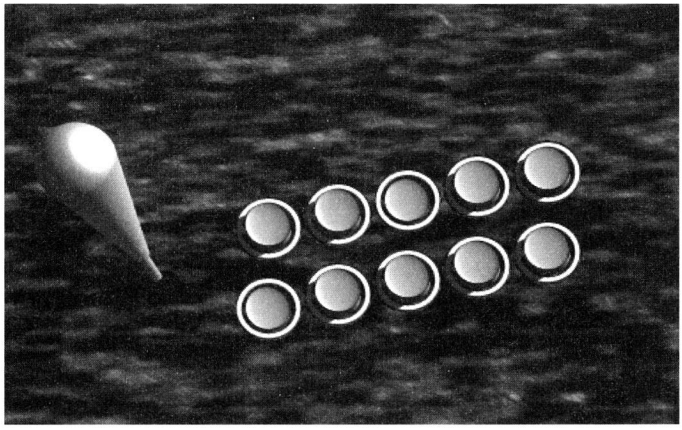


Abbildung 12.5: Das Joystick-Testprogramm

Der Anfang ist den bisherigen Beispielen sehr ähnlich. Nur dass jetzt der Joystick anstatt der Maus initialisiert wird.

```
// ------------ Main ----------------------
allegro_init();

if (!set_graphic_mode()) {
    allegro_message("Unable to set graphic mode!");
    exit(0);
}
install_timer();
install_keyboard();
install_joystick(JOY_TYPE_AUTODETECT);

BITMAP *doublebuffer = create_bitmap(SCREEN_W,
                            SCREEN_H);
BITMAP *bg           = load_bitmap("joybg.bmp"   ,
                            NULL);
BITMAP *btnNormal    = load_bitmap("button0.bmp" ,
                            NULL);
BITMAP *btnPressed   = load_bitmap("button1.bmp" ,
                            NULL);
BITMAP *joystick[9];
char buffer[128];
```

```
for (int a=0; a < 9; a++) {
    snprintf(buffer, 128, "joy%02d.bmp", a);
    joystick[a] = load_bitmap(buffer, NULL);
}
```

Eine zweite Neuerung ist die Art und Weise, wie die Bilder des Joysticks geladen werden. Der Dateiname der Bilder besteht aus »joy« gefolgt von der Bildnummer (zweistellig mit führender 0). Um nun diese neun Bilder zu laden, benutzen wir eine Schleife und `snprintf()`.

Die Funktion `snprintf()` funktioniert vom Prinzip her wie `printf()` abgesehen von zwei kleinen Unterschieden. Zum einen ist das Ziel der Ausgabe nicht der Bildschirm, sondern ein String, zum anderen kann man angeben, wie groß der resultierende String maximal sein darf.

```
snprintf(buffer, 128, "joy%02d.bmp", a)
```

Das Ziel ist `buffer` und es dürfen maximal 128 Zeichen in `buffer` geschrieben werden. Die Zeichenfolge `%02d` ist ein Platzhalter. Ein Platzhalter beginnt immer mit einem `%`. Die `02d` bedeutet:

- ✔ Mit führenden 0 ausgeben (0)
- ✔ Minimum 2 Zeichen (2)
- ✔ Dezimalzahl als Parameter (d)

Neben `%d` gibt es noch weitere Platzhalter, wie zum Beispiel `%s` (für Strings), `%f` (für Fließkommazahlen) und `%c` für einzelne Zeichen. Mehr zu den `printf()`-Parametern finden Sie in der C-Dokumentation und im Kapitel über Textausgabefunktionen in diesem Buch.

Nachdem nun alle Bilder geladen sind, wird nun das Layout der Buttons auf dem Schirm bestimmt.

```
int rows = joy[0].num_buttons / BUTTONS_PER_ROW;
if (joy[0].num_buttons % BUTTONS_PER_ROW > 0) {
    rows++;
}
if (rows < 3) {
    JOYSTICK_BTN_POS_Y += btnNormal->h;
}
```

Zuerst wird die Anzahl der Buttonreihen berechnet und dann ggf. die Position der oberen Reihe angepasst.

Und schon geht es in den Main-Loop:

```
// ------------ Main ----------------------
while (!key[KEY_ESC]) {
poll_joystick();
if (checkJoyDir(0, JOY_LEFT)) {
    if (checkJoyDir(0, JOY_UP)) {
        pos = 5;
    } else if (checkJoyDir(0, JOY_DOWN)) {

        pos = 8;
    } else {
        pos = 1;
    }
} else if (checkJoyDir(0, JOY_RIGHT)) {
    if (checkJoyDir(0, JOY_UP)) {
        pos = 6;
    } else if (checkJoyDir(0, JOY_DOWN)) {
        pos = 7;
    } else {
        pos = 2;
    }
} else if (checkJoyDir(0, JOY_UP)) {
    pos = 3;
} else if (checkJoyDir(0, JOY_DOWN)) {
    pos = 4;
} else {
    pos = 0;
}

// ------------ Display ----------------------
blit(bg, doublebuffer, 0, 0, panelX, panelY, bg->w, bg->h);
blit(joystick[pos], doublebuffer, 0, 0,
     panelX+JOYSTICK_POS_X,
     panelY+JOYSTICK_POS_Y,
     joystick[pos]->w,
     joystick[pos]->h);

int btn = 0;
for (int y=0; y < rows; y++) {
    for (int x=0; x < BUTTONS_PER_ROW && btn < joy[0].num_buttons;
    x++) {
        if (checkJoyButton(0, btn)) {
            draw_sprite(doublebuffer, btnPressed,
              panelX+JOYSTICK_BTN_POS_X+btnPressed->w*x,
```

```
                panelY+JOYSTICK_BTN_POS_Y + (btnPressed->h+10)*y - 10 *
              x);
          } else {
              draw_sprite(doublebuffer, btnNormal,
                panelX+JOYSTICK_BTN_POS_X+btnNormal->w*x,
                panelY+JOYSTICK_BTN_POS_Y+(btnPressed->h+10)*y - 10 *
              x);
          }
          btn++;
      }
  }
  vsync();
  blit(doublebuffer, screen, 0, 0, 0, 0, doublebuffer->w,
       doublebuffer->h);
  }
```

Die erste Aktion im Loop ist es, die Joystick-Informationen mit `poll_joystick()` aufzufrischen. Ohne diese Zeile ändert sich an den Werten im `joy`-Array nichts.

Die verschachtelten `if`-Abfragen ermitteln die Position des Knüppels. Wenn er nach links gedrückt wird, dann wird zusätzlich noch die vertikale Achse überprüft, um die Diagonalen mit einzuschließen. Das gleiche Verfahren wird für den Druck nach rechts angewandt. Wenn der Knüppel weder nach links noch nach rechts gedrückt wurde, dann wird die vertikale Achse geprüft. Da wir an dieser Stelle sicher sein können, dass weder nach links noch nach rechts gedrückt wird, erübrigen sich hier die Abfragen nach den Diagonalen. Wenn keine der beiden Achsen einen gültigen Wert hat, dann befindet sich der Joystickknüppel in der Ruheposition.

Zum Abfragen der Buttons wird die bereits oben vorgestellte Funktion `checkJoyButton()` benutzt. Gibt diese Funktion einen von 0 verschiedenen Wert zurück, dann ist der Button gedrückt und die `buttonPressed` Bitmap wird angezeigt. Ansonsten wird die `buttonNormal` Bitmap an diese Position gerendert. Der eigentliche Code für das Abfragen und Anzeigen ist ziemlich leicht zu verstehen. Nur der Code für die Berechnung der Anzeigeposition ist etwas komplizierter.

Der vollständige Code für dieses Programm befindet sich natürlich auf der beigelegten CD.

Zum Kompilieren benutzen Sie die Befehlszeile:

```
g++ -o prog12_5.exe prog12_5.cpp -lalleg
```

Ein virtuelles Joypad

Aus Sicht des Programmierers ist die Anzahl der wirklich vorhandenen Tasten normalerweise nicht von allzu großer Bedeutung. Was viel interessanter ist, ist die Anzahl der möglichen Aktionen des Spielers. Gehen wir einmal davon aus, dass wir für ein Spiel diese Aktionen brauchen:

- ✔ Bewegen
- ✔ Angriff
- ✔ Reden/Lesen/Benutzen
- ✔ Übersichtskarte
- ✔ Mitgeführte Gegenstände
- ✔ Spielmenü (Pause/Ende/Speichern)

Bewegen legen wir auf das Steuerkreuz. Die restlichen fünf Aktionen dann wohl auf die Buttons 1-5. Wie diese Aktionen nun auf die Buttons gelegt werden, ist vom Prinzip her egal, da es eh nicht vorhersehbar ist, wie die Buttons auf dem Joypad des Spielers angeordnet sind. Wir können nicht mal sicher sein, dass das Pad überhaupt fünf Buttons hat.

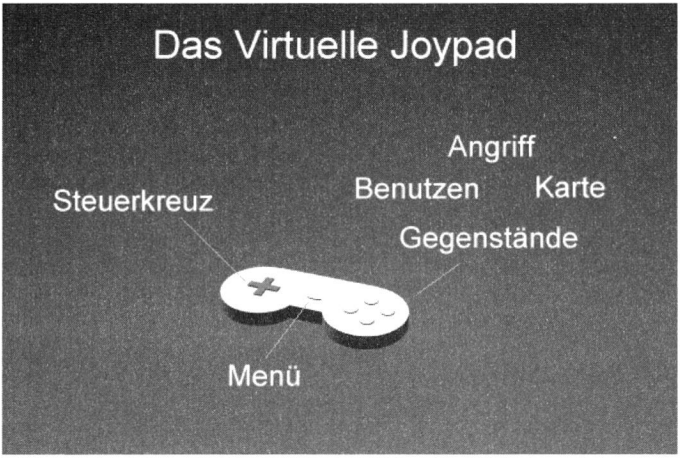


Abbildung 12.6: Das virtuelle Joypad

Hat das Joypad nur vier Buttons, dann muss eine der Funktionen, vorzugsweise das Menü, auf eine Taste am Keyboard gelegt werden.

Hat der Spieler überhaupt kein Joypad, dann sollten alle Funktionen auf sinnvollen Tasten liegen.

Die Lösung dieses Problems ist einfach: Unser virtuelles Joypad reagiert sowohl auf das Keyboard als auch auf das Joypad (wenn denn eines angeschlossen ist).

Um das virtuelle Pad auch später einfach einsetzen zu können, werden wir es in eine eigene Klasse auslagern.

```
class Joypad {


public:
    // Das Steuerkreuz
    enum {
        UP    = 0,
        DOWN  = 1,
        LEFT  = 2,
        RIGHT = 3
    };
    // Die Knöpfe
    enum {
        ACTION    = 4,
        USE       = 5,
        MAP       = 6,
        INVENTORY = 7,
        MENU      = 8
    };

    // Die anzahl der knöpfe (inklusive steuerkreuz)
    enum {
        BUTTON_COUNT = 9;
    };
    // die eigentlichen werte
    int button[BUTTON_COUNT];

    // Der Konstruktor
    Joypad();
    // Die poll Methode füllt das button array mit
    // gültigen Werten.
    void poll();
};
```

Die `poll()` Methode erledigt die gesamte Arbeit, fragt Keyboard und Joystick ab und füllt dann die Variablen entsprechend.

Normalerweise sollte man Variablen nicht als `public` deklarieren, sondern nur über Methoden der Klasse auf sie zugreifen. In diesem Fall machen wir eine Ausnahme, um den Zugriff auf diese Werte zu erleichtern.

Die einzelnen Aktionen werden über mehrere `enum` definiert. Auf diese Weise kann man Klassenkonstanten innerhalb der Klasse selbst definieren. Der Zugriff auf einen Wert wird dann in etwa so aussehen:

```
Joypad joy;
joy.poll();
if (joy.button[Joypad::UP]) {
    // Und hier die Aktion
}
```

Da die einzelnen Joystickknöpfe alle in einem Array definiert sind, kann man in der `poll()` Methode die Werte in einer Schleife zuweisen. Zu diesem Zweck definieren wir erst einmal die entsprechenden Tastaturkommandos.

```
int key[BUTTON_COUNT];
```

Im Konstruktor werden dann dem `key` Array Defaultwerte zugewiesen und in der `poll()` Methode wird das `button` Array dem entsprechend gefüllt.

```
Joypad::Joypad() {
    int defaultValues[BUTTON_COUNT] = {
        KEY_UP,
        KEY_DOWN,
        KEY_LEFT,
        KEY_RIGHT,
        KEY_SPACE,
        KEY_ENTER,
        KEY_M,
        KEY_I,
        KEY_ESC
    };
    for (int a=0; a < BUTTON_COUNT; a++) {
        key[a] = defaultValues[a];
    }
}
```

```
void Joypad::poll() {
    poll_joystick();
    for (int a=0; a < BUTTON_COUNT; a++) {
        if (::key[key[a]]) {
            button[a] = 1;
        } else {
            button[a] = 0;
        }
    }
}
```

Hier gibt es eine Besonderheit: Da die Klasse eine Variable `key[]` definiert, wird damit das globale `key` Array von Allegro überdeckt. Um wieder auf das globale Array zugreifen zu können, müssen wir dieses explizit durch das Voranstellen von `::` angeben.

Bisher fragen wir aber nur die Tastatur ab, Joystick-Eingaben bleiben unberücksichtigt. Aber das ändern wir nun.

Jeder virtuelle Button kann entweder einer Achse oder einem Button entsprechen. Um einen Knopf auf dem Joystick zu beschreiben ist es ausreichend den Index des Knopfes und des entsprechenden Joysticks anzugeben. Bei einer Achse brauchen wir den Joystick, die Achse und ob es der minimale oder maximale Ausschlag ist. Diese letzte Unterscheidung (`d1` oder `d2`) entspricht in etwa dem Button Index bei den Knöpfen.

```
Joystick 0, Stick 0, Achse 0, Knopf 0 =
    joy[0].stick[0].axis[0].d1
Joystick 0, Stick 0, Achse 0, Knopf 0 =
    joy[0].axis[0].d2
```

Wir können also mit vier Variablen (`device, stick, axis, button`) alle Fälle abbilden. Um nun zwischen einer Achse und einem Knopf unterscheiden zu können, brauchen wir einen Wert, der »ungültig« bedeutet.

```
class JoystickInfo {
public:
    enum {
        INVALID = -1
    };
    int device, stick, axis, button;

    JoystickInfo() {
        set(INVALID, INVALID,INVALID,INVALID);
    }
    JoystickInfo(int device, int button) {
        set(device, button);
```

```
    }
    JoystickInfo(int device, int stick, int axis, int button) {
        set(device, stick, axis, button);
    }

    void set(int device, int button) {
        this->device = device;
        this->stick  = INVALID;
        this->axis   = INVALID;
        this->button = button;
    }
    void set(int device, int stick, int axis, int button) {
        this->device = device;
        this->stick  = stick;
        this->axis   = axis;
        this->button = button;
    }
};
```

Um nun den zweiten Button am ersten Joystick zu beschreiben reicht:

```
JoystickInfo button(0, 1);
```

Wollen wir den d2 Wert an der 2ten Achse des 1sten Knüppels am ersten Joystick (also den »nach unten«Wert):

```
JoystickInfo down(0,0, 1, 1);
```

Wir können nun auf diese Weise auch eine Default Joystickbelegung speichern.

```
JoystickInfo joy[BUTTON_COUNT];
```

Diese wird im Konstruktor genauso zugewiesen wie beim key Array. Allerdings muss man hier ein paar Checks mehr machen, um sicher zu gehen, dass auch alle Werte im gültigen Rahmen sind.

```
Joypad::Joypad() {
    int defaultValues[BUTTON_COUNT] = {
        KEY_UP,
        KEY_DOWN,
        KEY_LEFT,
        KEY_RIGHT,
        KEY_SPACE,
        KEY_ENTER,
        KEY_M,
        KEY_I,
```

```
        KEY_ESC
    };
    JoystickInfo defJoy[] = {
        // Richtungen
        JoystickInfo(0,0,1,0),
        JoystickInfo(0,0,1,1),
        JoystickInfo(0,0,0,0),
        JoystickInfo(0,0,0,1),
        // Knöpfe
        JoystickInfo(0,0),
        JoystickInfo(0,1),
        JoystickInfo(0,2),
        JoystickInfo(0,3),
        JoystickInfo(0,4),
    };
    for (int a=0; a < BUTTON_COUNT; a++) {
        key[a] = defaultValues[a];
        joy[a].stick = JoystickInfo::INVALID;
        if (num_joysticks > defJoy[a].device) {
            if (defJoy[a].stick != JoystickInfo::INVALID) {
                if (defJoy[a].stick <
::joy[defJoy[a].device].num_sticks) {
                    if (defJoy[a].axis <
::joy[defJoy[a].device].stick[defJoy[a].stick].num_axis) {
                        switch (defJoy[a].button) {
                            case 0:
                            case 1:
                                joy[a].device = defJoy[a].device;
                                joy[a].stick  = defJoy[a].stick;
                                joy[a].axis   = defJoy[a].axis;
                                joy[a].button = defJoy[a].button;
                            break;
                        }
                    }
                }
            } else if (::joy[defJoy[a].stick].num_buttons >
            defJoy[a].button) {
                joy[a].stick  = defJoy[a].device;
                joy[a].button = defJoy[a].button;
                joy[a].stick  = JoystickInfo::INVALID;
                joy[a].axis   = JoystickInfo::INVALID;
            }
        }
    }
}
```

Wenn man nun den Joystick mit diesen Informationen abfragt, dann sieht das so aus:

```
if (joy[a].device != JoystickInfo::INVALID) {
  // Button oder Richtung?
  if (joy[a].stick != JoystickInfo::INVALID) {
    // Richtung
    if (joy[a].button == 0) {
      button[a] = ::joy[joy[a].device].stick[
                    joy[a].stick].axis[joy[a].axis].d1;
    } else {
      button[a] = ::joy[joy[a].device].stick[
                   joy[a].stick].axis[joy[a].axis].d2;
    }
  } else {
    // Button
    button[a] =
        ::joy[joy[a].device].button[joy[a].button].b;
 }
} else {
  button[a] = 0;
}
```

Während dies sicherlich eine funktionierende Lösung ist, bringt sie doch einen ziemlichen Overhead mit sich. Selbst im besten Fall benötigen wir zwei `if`-Abfragen um einen Wert zu lesen. Die verschachtelten Arrays vergrößern die Zugriffszeit und machen den Code schwer lesbar. Also muss eine bessere Lösung her.

Rein prinzipiell fragen wir immer nur ein `int` ab. Nur die Position dieses `int`s innerhalb des `joy`-Arrays ist immer unterschiedlich. Wir können also, anstatt uns jedes Mal durch die gesamte Struktur zu hangeln, einen Zeiger auf den tatsächlichen Wert speichern. Die `JoystickInfo` Klasse brauchen wir auch weiterhin, da sich die Position des `joy`-Arrays bei jedem Programmstart ändert.

Fügen wir also ein Array von `int`-Zeigern in die `Joypad` Klasse ein:

```
int* joyState[BUTTON_COUNT];
```

Und setzten diesen Wert dann im Konstruktor nachdem wir die Joystick-Informationen geprüft haben. Im Falle eines ungültigen Wertes setzen wir den Zeiger auf `NULL`.

Die wunderbare Welt der Kapselung. Obwohl wir die Arbeitsweise des Joypads geändert haben, können wir immer noch genauso darauf zugreifen wie bisher. Dies hat einen weiteren Vorteil: Wenn Sie sich noch immer etwas unsicher mit Zeigern fühlen, dann können Sie diese getrost vergessen und wie bisher auch auf das `button` Array der `Joypad` Klasse zugreifen.

Nun können wir den Joystick genauso einfach abfragen wie wir bisher die Tastatur abgefragt haben:

```
void Joypad::poll() {
    poll_joystick();
    for (int a=0; a < BUTTON_COUNT; a++) {
        if (::key[key[a]]) {
            button[a] = 1;
        } else {
            if (joyState[a] != NULL) {
                button[a] = *joyState[a];
            } else {
                button[a] = 0;
            }
        }
    }
}
```

Wir könnten nun auch auf recht einfache Weise die Werte der `key` und `joy` Arrays anpassen beziehungsweise vom Spieler anpassen lassen. Allerdings verschieben wir das etwas nach hinten.

Testprogramm

Eine angepasste Version des Joystick-Testprogramms finden Sie auf der Buch-CD (`prog12_6.cpp`). Beachten Sie bitte, dass Sie sowohl das `joypad.cpp` als auch das Beispielprogramm `prog12_6.cpp` kompilieren müssen.

```
g++ -o prog12_6.exe prog12_6.cpp joypad.cpp -lalleg
```

Sie können natürlich auch die beiden Dateien getrennt kompilieren und dann zu einer ausführbaren Datei linken:

```
g++ -o prog12_6.o -c prog12_6.cpp
g++ -o joypad.o -c joypad.cpp
g++ -o prog12_6.exe prog12_6.o joypad.o -lalleg
```

13 Ein erstes Spiel

Die Grundlagen sind gelegt: Timing, Laden und Anzeigen von Bildern und Sounds. Abfrage von Benutzereingaben über Tastatur, Maus und Joystick. Nun wird es Zeit all dies zusammenzuführen und ein erstes Spiel zu erstellen. Neben der Planung und Umsetzung wird auch gezeigt, wie man dieses kleine Spiel als Minispiel in ein Rollenspiel integrieren kann.

Am Anfang steht die Planung

Der erste Schritt, wenn man ein Spiel programmiert, ist immer die Planung. Was will man erreichen? Welchen Umfang soll das Spiel haben? Welche Features sollen implementiert werden?

Diese Planung erfüllt einen einfachen Zweck: Zum einen hat man beim Programmieren einen Leitfaden. Zum anderen weiß man, wann man fertig ist. Der letzte Punkt klingt etwas seltsam, ist aber einer der häufigsten Gründe, warum Spiele von Hobby-Programmierern nicht beendet werden. Das Spiel ist fertig, macht Spaß und sieht gut aus. Und dann schlägt der »Ich-könnte-noch«-Faktor gnadenlos zu. Schritt für Schritt werden Sachen hinzugefügt, die eigentlich gar nicht geplant waren. Scheinbar kleine Änderungen ziehen Probleme nach sich, an die man im ersten Moment gar nicht dachte. Und einige Zeit später wird das Projekt auf Eis gelegt, um all diese Features in der nächsten Version von vorne herein zu berücksichtigen.

Natürlich geht das Spiel beim nächsten Projekt von vorne los. Wieder kommen neue Ideen hinzu und verzögern die Fertigstellung des Spieles.

Es gibt ein einfaches Mittel gegen dieses Problem: Planung.

Das Design-Dokument

Wie bereits im ersten Teil des Buches erwähnt, ist das Design-Dokument das zentrale Element der Spielplanung. Es sollte die Idee, den Spielverlauf und das grundlegende »Look and Feel« des Spieles enthalten. Alle wichtigen Features des Spieles müssen enthalten sein, ebenso wie ein

»Walkthrough« durch das Spiel. Ein Walkthrough ist eine Auflistung aller im Spiel vorkommenden Elemente. Angefangen beim Titelbildschirm über das eigentliche Spiel bis hin zum »Game Over«.

Schritt 1: Der Schlüsselsatz

Der Schlüsselsatz fasst das gesamte Spielprinzip in einem (oder ein paar wenigen) Sätzen zusammen:

»Das Spiel ist ein Gedächtnispuzzle, in dem der Spieler sich immer länger werdende Folgen von Farben und Tönen merken und dann nachspielen muss«.

Ich gebe zu, dies ist nicht besonders innovativ. Aber für den Anfang ist es durchaus ausreichend.

Schritt 2: Eine genauere Spielbeschreibung

Der Spieler sieht eine Scheibe mit vier farbigen Segmenten. Alle Segmente sind gleich groß und haben unterschiedliche Farben: Rot, Grün, Gelb und Blau. Diese Segmente leuchten in einer zufälligen Reihenfolge nacheinander auf. Bei jedem Aufleuchten wird ein akustisches Signal gegeben. Der Spieler muss nun die Folge in der gleichen Reihenfolge nachspielen.

Ist er erfolgreich, so wird die Folge um ein weiteres Element erweitert. Macht er einen Fehler, so wird dies mit einem entsprechenden Text und akustischer Untermalung angezeigt. Der Spieler darf nicht mehr als drei Fehler machen, sonst ist das Spiel beendet.

Schritt 3: Planung der einzelnen Bildschirme

Das Spiel hat unterschiedliche Screens:

- ✔ Das Titelbild
- ✔ Das Spielfeld
  - ✔ Anzeige: Neue Folge
  - ✔ Anzeige: Spieler am Zug
  - ✔ Anzeige: Fehler
- ✔ Der Game Over Screen

Das Titelbild

Anzeige des Spielnamens und der folgenden Optionen:

- ✔ Spielstart
- ✔ Spielende

Das Spiel wird mit Esc beendet und durch Drücken einer beliebigen anderen Taste gestartet.

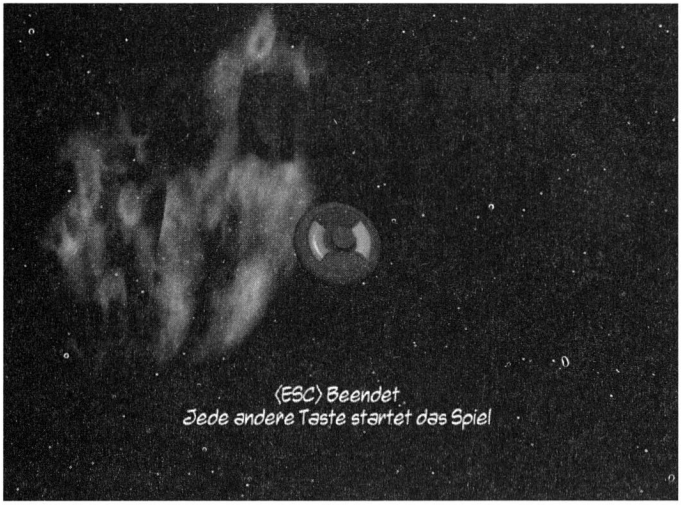


Abbildung 13.1: Das Titelbild

Das Spielfeld

Im Zentrum des Spieles ist die Scheibe zu sehen. Jede farbige Fläche ist klar einem Bereich zugeordnet: Grün ist oben. Blau ist rechts. Gelb ist unten. Rot ist auf der linken Seite (siehe Abbildung 13.2).

Anzeige neue Folge

Bevor der Computer die Folge spielt, wird für einen kurzen Moment ein Text in der Mitte des Schirms angezeigt »Achtung«. Der Text verschwindet und die Folge wird abgespielt.

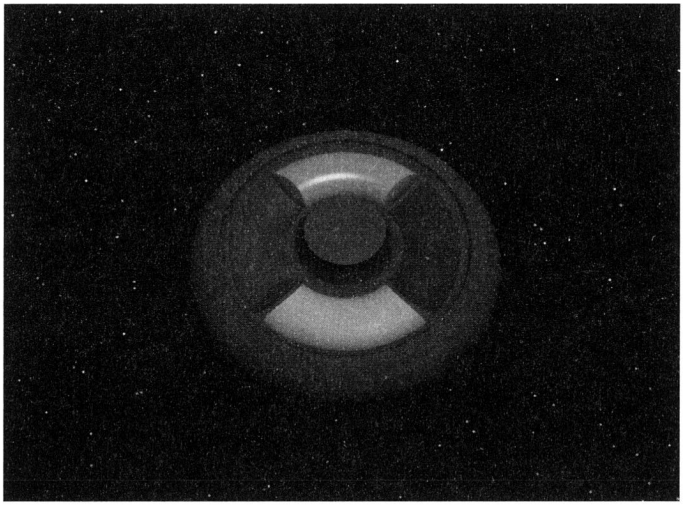


Abbildung 13.2: Das Spielfeld

Spieler am Zug

Nachdem die Folge beendet ist, wird ein Text »Und jetzt Sie...« zentriert auf dem Schirm angezeigt. Nun kann der Spieler seine Eingaben machen.

Fehler

Macht der Spieler beim Nachspielen der Folge einen Fehler, so wird der Text »Oups!« zentriert dargestellt. Dann spielt der Computer die Folge noch einmal und der Spieler erhält eine weitere Chance.

Spielende

Macht der Spieler drei Fehler, so wird das Spiel beendet. Ein Schriftzug »Game Over« wird zentriert angezeigt. Nach einer kurzen Pause geht es weiter zum Titelbildschirm (siehe Abbildung 13.3).

Normalerweise werden Sie anfangs noch keine Screenshots haben – aber eine Skizze des geplanten Bildschirms tut es auch.

Nachdem nun die generelle Planung steht, kann man mit der Planung der Umsetzung beginnen.

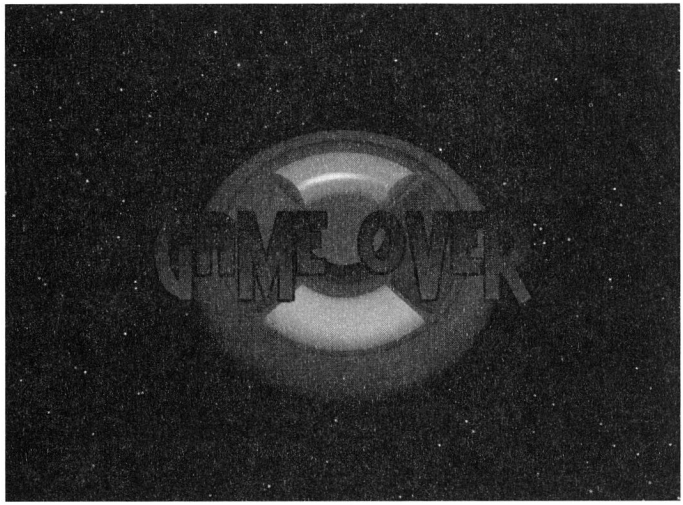


Abbildung 13.3: Der Game-Over-Schirm des Spieles

Umsetzen des Konzepts

Um es etwas einfacher zu machen, werden wir in diesem Beispiel jede Menge Speicherplatz verschenken: Alle Grafiken sind bildschirmfüllend. Diese Lösung ist zwar nicht ideal, aber bei einem solchen Projekt nicht hinderlich. Sie können die Grafiken und die Position an der diese angezeigt werden, auch zu einem späteren Zeitpunkt optimieren.

Das Grundgerüst

Die Grundlage für dieses Spiel bildet der bisher verwendete Code. Wir werden die Klasse für das virtuelle Joypad benutzen und den bereits erarbeiteten Timer Code wiederverwenden.

Wie wir bereits aus dem Design-Dokument entnehmen können haben wir drei große Bereiche:

- ✔ Titelbild
- ✔ Spiel
- ✔ Game Over

Jeder Bereich wird in eine eigene Datei ausgelagert, damit der Code überschaubar bleibt. Na ja, ich gebe zu, ein weiterer Grund dafür mehrere Dateien zu benutzen ist auch, Makefiles noch einmal an einem überschaubaren Beispiel durchzugehen.

Neben den Dateien für diese drei Bereiche haben wir noch die Joypad Files (`joypad.h` und `joypad.cpp`), die Sie einfach in einen neuen Ordner kopieren. Wir brauchen auch noch eine Datei, um die ganzen Einzelteile zu verbinden. Damit kommen wir auf diese Liste von Dateien:

Name	Beschreibung
`iqchallenge.cpp`	Das Hauptprogramm. Verbindet alle anderen Teile und stellt den Timing Code, das Joypad und den DoubleBuffer zur Verfügung.
`iqchallenge.h`	Definiert die von außen zugänglichen Funktionen und Variablen
`title.cpp`	Der Titelschirm
`title.h`	Macht die Titelfunktion nach außen zugänglich.
`game.cpp`	Das eigentliche Spiel
`game.h`	Macht die Spielfunktion nach außen zugänglich.
`gameover.cpp`	Der Game-Over-Schirm
`gameover.h`	Header Datei für das Spielende
`joypad.cpp`	Das virtuelle Joypad
`joypad.h`	Der entsprechende Header
`makefile`	Das Makefile, um aus all diesen Dateien ein Spiel zu erstellen.

Tabelle 13.1: Die Liste aller Quellcode-Dateien

Legen Sie ein neues Verzeichnis an und kopieren Sie joypad.cpp und joypad.h aus dem letzten Kapitel in dieses neue Verzeichnis.

Erstellen Sie eine neue Datei namens »iqchallenge.cpp« und füllen Sie sie mit den grundlegenden Funktionen.

Diese Datei finden Sie auch auf der CD unter dem Namen iqchallenge_Schritt1.cpp.

Die Datei können Sie auch als allgemeinen Ausgangspunkt für weitere Spiele betrachten.

```
#include <allegro.h>
#include <time.h>


#include "iqchallenge.h"

BITMAP   *doubleBuffer = NULL;
Joypad   *joypad;

// Die Timer Variablen
// ----------------------------------------------
volatile int timerCounter = 0;
static void timerCounterUpdater() {
    timerCounter++;
}
END_OF_STATIC_FUNCTION(timerCounterUpdater);

// Die normalen Funktionen
// ---------------------------------------------------
void fatalError(char *str) {
    allegro_message(str);
    exit(0);
}

int setGfxMode(int mode, int width, int height) {
    static int colorDepths[] = { 16, 24, 15, 32};
    int a;

    for (a=0; a < 4; a++) {
        set_color_depth(colorDepths[a]);
        if (set_gfx_mode(mode,width,height,0,0) >= 0) {
            return 1;
```

```
        }
    }
    return 0;
}

void show() {
    blit(doubleBuffer, screen, 0,0,0,0, #
         doubleBuffer->w, doubleBuffer->h);
}

void init() {
    // Zufallszahlen Generator initialisieren
    srand(time(NULL));
    // Allegro und Grafik, Timer, Sound und
    // Eingabemedien initialisieren
    allegro_init();
    if (!setGfxMode(GFX_AUTODETECT_FULLSCREEN, 640, 480)) {
        if (!setGfxMode(GFX_AUTODETECT_WINDOWED, 640, 480)) {
            fatalError("Unable to set a graphics mode");
        }
    }
    install_timer();
    install_keyboard();
    install_joystick(JOY_TYPE_AUTODETECT);
    install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, NULL);

    /* Den DoubleBuffer erstellen */
    doubleBuffer = create_bitmap(SCREEN_W, SCREEN_H);
    if (!doubleBuffer) {
        fatalError("Unable to create double buffer");
    }

    /* Timer Funktionen und Variablen locken */
    LOCK_FUNCTION(timerCounterUpdater);
    LOCK_VARIABLE(timerCounter);
    /* Wir wollen 30 Bilder pro Sekunde */
    install_int_ex(timerCounterUpdater,
                   BPS_TO_TIMER(30));
    // Joypad initialisieren
    joypad = new Joypad();
}

// Gibt die in init() angelegten Ressourcen
// wieder frei
```

```
void done() {
    destroy_bitmap(doubleBuffer);
}

// Das Hauptprogramm
// ------------------------------------------------
int main(int argc, char** argv) {
    int gameOver = FALSE;

    init();

    while (!gameOver) {
        gameOver = key[KEY_ESC];
    }

    done();
} END_OF_MAIN();
```

Hier eine kurze Übersicht über die Funktionen

Name	Bedeutung
`fatalError`	Gibt eine Fehlermeldung aus und beendet dann das Programm.
`setGfxMode`	Setzt den angegebenen Grafikmodus. Kann der Modus nicht gesetzt werden, so wird `FALSE` zurückgegeben, ansonsten `TRUE`.
`show`	Zeigt den Double Buffer auf dem Schirm an.
`init`	Sorgt dafür, dass der Grafikmodus gesetzt wird, Sound, Keyboard, Joystick und Timer funktionieren und legt den Double Buffer an.
`done`	Gibt die in `init` erzeugten Ressourcen wieder frei.
`main`	Das Hauptprogramm

Tabelle 13.2: Übersicht über die Funktionen im Hauptprogramm

Es gibt einige kleinere Änderungen im Code, verglichen zu denen in den vorherigen Kapiteln. Zum einen haben wir nun eine Funktion `fatalEr-`

ror(), die nach einer Fehlermeldung das Programm beendet. Diese Funktion wird benutzt um einen Fehler anzuzeigen, wenn kein Grafikmodus gesetzt werden kann. Wir werden sie später benutzen, um das Programm zu beenden, wenn eine Datei (Grafik oder Sound) nicht geladen werden kann.

Die joypad Variable ist nun ein Zeiger. Dies hat einen sehr einfachen Grund: Globale Variablen werden direkt bei Programmstart erzeugt. Zu diesem Zeitpunkt ist Allegro aber noch nicht initialisiert, d.h. die im Konstruktor verwendeten Allegro-Aufrufe bleiben erfolglos. Wir müssen dieses Objekt also erzeugen, nachdem Allegro einsatzbereit ist. Aus diesem Grund wird das Objekt in init() dynamisch erzeugt und in done() wieder freigegeben.

Der nächste Schritt ist, eine Header Datei für die global zugänglichen Variablen und Funktionen zu erstellen. Die globalen Variablen sind der Double Buffer, der Timer und das Joypad. Bei den Funktionen geben wir nur fatalError() und show() nach außen weiter.

```
#ifndef IQ_CHALLENGE_HEADER
#define IQ_CHALLENGE_HEADER
#include  <allegro.h>
#include "joypad.h"

// Global zugängliche Variablen
extern BITMAP*      doublebuffer;
extern volatile int timerCounter;
extern Joypad       *joypad;

// Von außen aufrufbare Funktionen
void fatalError(char *str);
void show();

#endif
```

Noch ein Schritt und wir haben das Gröbste hinter uns: Das Makefile.

Das Makefile

Wir werden uns bei diesem Makefile in erster Linie auf die impliziten Regeln verlassen. Auf diese Weise bleibt das Makefile übersichtlich kurz.

```
CPP=g++
CPPFLAGS=-mwindows
#Windows Variante
LIBS=-lalleg
#Linux Variante
#LIBS=- 'allegro-config --libs'

EXE=iqchallenge
OBJECTS= ${EXE}.o joypad.o

${EXE}: ${OBJECTS}
    ${CPP} -o ${EXE} ${OBJECTS} ${CFLAGS} ${LIBS}
```

Stellen Sie sicher, dass Sie im Makefile mit ⇥ einrücken.

Die Variable EXE enthält den Namen des endgültigen Programms. In OBJECTS listen wir die Namen der einzelnen Objektdateien auf. Solange alle Objektdateien so heißen wie der Quellcode (abgesehen von der Endung), können wir neue Dateien hinzufügen, indem wir nur diese Zeile ändern.

Die Regel zum Erstellen der EXE braucht die in OBJECTS aufgezählten Dateien. Um diese zu erzeugen, benutzt Make die intern definierten Regeln, um aus .cpp-Dateien .o-Dateien herzustellen. Der Vorteil für uns ist, dass wir uns jede Menge monotoner Tipparbeit sparen.

Der Grundstein ist gelegt. Jetzt können wir uns an unserem Design-Dokument durch das Spiel hangeln und nacheinander alle Funktionen implementieren.

Das Titelbild

Das Titelbild ist eine recht einfache Funktion. Wir zeigen ein Bild an und warten auf einen Tastendruck. Ist die gedrückte Taste , dann soll sich das Programm beenden. Ist es eine andere Taste, dann beginnt ein neues Spiel.

Damit der Titel aus dem Hauptprogramm aufgerufen werden kann, müssen wir ein Header File erzeugen:

```
#ifndef TITLE_HEADER
#define TITLE_HEADER

// Gibt TRUE zurück falls ein neues Spiel
// starten soll. FALSE bei Programmende
```

```
int title();

#endif
```

Diese Datei (`title.h`) wird sowohl im Hauptprogramm als auch in `title.cpp` inkludiert.

Da `title()` eine wirklich sehr simple Funktion ist, präsentiere ich Ihnen gleich den gesamten (kurzen) Quellcode und gehe dann auf die einzelnen Schritte ein.

```
#include "iqchallenge.h"
#include "title.h"

// Gibt TRUE zurück wenn ein neues Spiel gestartet
// werden soll, FALSE bei Programm Ende
int title() {
    BITMAP *bg = load_bitmap("title.tga", NULL);
    if (!bg) {
        fatalError("Bild \"title.tga\" konnte nicht "
                   "gefunden werden\n"
                   "Programm wird beendet.");
    }
    blit(bg, doubleBuffer, 0, 0, 0, 0, bg->w, bg->h);
    // Wir brauchen das Bild nicht mehr
    // Am doubleBuffer ändert sich die Funktion
    // über nichts
    destroy_bitmap(bg);

    // Sicherstellen, dass keine Tasten
    // im Puffer stehen
    clear_keybuf();
    while (!keypressed()) {
        show();
    }
    // Taste auslesen
    int keyValue = readkey();
    if ((keyValue >> 8) == KEY_ESC) {
        // Programm Ende
        return FALSE;
    }
    // Sicherheitshalber den Puffer löschen
    clear_keybuf();

    // Neues Spiel beginnen
    return TRUE;
}
```

Zu Beginn der Funktion wird das Titelbild geladen. Konnte die Datei nicht gefunden werden, wird eine Fehlermeldung ausgegeben. Um innerhalb des Textes Anführungszeichen benutzen zu können, muss diesen ein Backslash \ vorangestellt werden. Ein Zeilenumbruch wird durch `\n` eingeleitet. Folgen mehrere Strings nur durch Leerzeichen getrennt aufeinander, so hängt der Compiler diese automatisch aneinander.

Nachdem das Bild erfolgreich geladen wurde, wird es sofort auf den Double Buffer geblittet. Ab diesem Zeitpunkt brauchen wir das eigentliche Bild nicht mehr. Also wird es in der nächsten Zeile freigegeben.

Nun beginnt das Warten auf einen Tastendruck. Um nicht auf bereits im Tastaturpuffer stehende Eingaben zu reagieren, wird erst der Puffer gelöscht. Die nun folgende Schleife wartet auf einen Tastendruck und zeigt währenddessen den Double Buffer auf dem Schirm an.

Wird eine Taste gedrückt, dann überprüfen wir den Scancode, ob es Esc war. Der Scancode befindet sich in den oberen acht Bits, also verschieben wir diese um acht Stellen nach rechts. Nun können wir mit `KEY_ESC` vergleichen. Ist der ermittelte Scancode der für die Escape-Taste geben wir `FALSE` zurück.

Ansonsten wurde offensichtlich nicht Esc gedrückt. Wir löschen den Tastaturpuffer (nur um sicher zu gehen) und geben `TRUE` zurück.

Ich hatte ja bereits gesagt: Die Titelbildfunktion ist sehr einfach zu implementieren. Damit es auch kompiliert wird, müssen wir nun noch das Makefile anpassen. Ändern Sie die `OBJECTS` Zeile wie folgt ab:

```
OBJECTS= ${EXE}.o joypad.o title.o
```

Dadurch wird sichergestellt, dass die Datei kompiliert wird. Um nun das Titelbild auch aufzurufen, brauchen wir zwei kleine Änderungen am Hauptprogramm.

Zum einen muss `title.h` inkludiert werden. Zum anderen müssen wir die Funktion auch aufrufen.

```
while (!gameOver) {
    if (!title()) {
        gameOver = TRUE;
    }
}
```

Den Quellcode bis hier finden Sie auf der CD unter dem Namen `iqchallenge_Schritt2.cpp`.

Das Spiel

Bevor wir mit der eigentlichen Programmierung beginnen, müssen wir uns erst überlegen, auf welche Weise wir die Daten speichern wollen. Wir haben vier verschiedene Zustände (Rot, Grün, Blau und Gelb). Um diese zu speichern, brauchen wir nur zwei Bits. Also könnten wir vier Zustände in Byte packen. Allerdings erkaufen wir uns diese Platzersparnis mit einem höheren Aufwand beim Zugriff auf die Daten. Ein weiterer Vorteil ist, dass wir dann die STL-Klassen ohne jedes Problem einsetzen können.

```
#include <vector>

using namespace std;
typedef vector<char> CharVector;
typedef CharVector::iterator CharVectorIterator;
```

Jetzt können wir ein dynamisches Array von chars erstellen, dessen Speicherplatz bei Bedarf erweitert wird.

Vector oder Liste? Vom Prinzip her könnte man beide STL-Klassen verwenden. Die Wahl fiel jedoch aus einem bestimmten Grund auf den Vector: Eine Liste speichert für jedes Datenelement zwei Zeiger. Einen zum vorherigen und einem zum folgenden Element. Ein Zeiger ist jedoch vier Bytes groß. Aus diesem Grund würde ein char-Element einen Verwaltungsaufwand in acht-facher Größe verursachen.

Wenn wir die STL-Klassen benutzen, müssen wir unser Makefile leicht anpassen, da die Windows-Version des GNU Compilers beim Inkludieren der STL-Header automatisch auch die Windows-System-Header inkludiert. Und da dies nicht erwünscht ist (und zu Fehlern führt), schieben wir dieser Sache einen Riegel vor. Ändern Sie bitte die `CPPFLAGS` Deklaration im Makefile wie folgt ab:

```
CPPFLAGS=-mwindows -D__GTHREAD_HIDE_WIN32API
```

Damit nicht bei jedem Einfügen eines neuen Elements auch neuer Speicher angelegt werden muss, reservieren wir Speicher für 128 Einträge. Dies sollte für den durchschnittlichen Spieler durchaus ausreichen.

Nachdem damit die Speicherstruktur für das Spiel bereitgestellt ist, können wir nun anfangen es mit Daten zu füllen. Zu Beginn besteht die Folge aus drei Signalen, also fügen wir drei zufällig erzeugte Zahlen in den Vector ein.

```
// Die ersten 3 Elemente
for (int a=0; a < 3; a++) {
    memory.push_back(rnd(4));
}
```

Die Methode `rnd()` ist eine Hilfsfunktion, die eine Zahl zwischen 0 und dem übergebenen Parameter zurückliefert.

```
int rnd( int n ){
    return (int)
        (n * ( (double)rand() /
              (double) (RAND_MAX + 1.0 )
             )
        );
}
```

Der Grund für diese etwas komplizierte Variante ist recht einfach: Die meisten Zufallszahlen-Generatoren sind in den unteren Bits deutlich weniger zufällig als in den oberen. Und gerade bei den kleinen Werten, die wir für dieses Spiel benötigen, ist dieses Problem sehr deutlich zu sehen.

Jetzt sind wir eigentlich für das Spiel soweit gerüstet. Nun müssen wir nur noch die Grafiken und den Sound laden und dann kann es losgehen.

Spielablauf

Zuerst wird die »Achtung«-Grafik angezeigt. Dann spielt der Rechner einem die Folge vor. Danach ist der Spieler am Zug (»Und-jetzt-Sie«-Grafik) und spielt die Folge nach. Macht er hierbei einen Fehler, dann wird sein »Mögliche-Versuche«-Zähler um eins verringert und die Sequenz wird wiederholt. Erreicht der Zähler 0, ist das Spiel zu Ende. Schafft der Spieler die Sequenz, so wird ein weiterer Wert angehängt, und es beginnt von neuem.

Aus dieser Beschreibung wird deutlich, dass es in diesem Spiel fünf Zustände gibt:

- Achtung-Status: Gleich wird die Folge vom Computer gespielt.
- Vorgabe-Status: Der Computer spielt die Folge die es nachzuspielen gilt.
- Bereit-Status: Der Spieler ist gleich an der Reihe.
- Spiel-Status: Der Benutzer spielt das eigentliche Spiel.
- Game-Over-Status: Das Spiel ist zu Ende, der Spieler hat verloren.

Je nach dem aktuellen Zustand des Spieles werden andere Grafiken angezeigt, auf Eingaben reagiert (oder auch nicht).

Hat ein Program:m eine Reihe fest definierter Zustände und lassen sich die Übergänge zwischen diesen Zuständen in Regeln fassen, so spricht man von einer ***Finite State Machine*** (FSM) oder von einem endlichen Automaten. Dieses Konzept ist sehr vielseitig einsetzbar. Neben dem eigentlichen Programmablauf lassen sich so auch Routinen für Computer-Gegner erstellen (mit Zuständen wie Angriff, Abwehr, Flucht, Stellung halten etc).

Für die Zustände im Spiel wurden folgende Konstanten definiert:

```
const int STATE_READY_TO_WATCH  = 0;
const int STATE_WATCH           = 1;
const int STATE_READY_TO_PLAY   = 2;
const int STATE_PLAY            = 3;
const int STATE_GAME_OVER       = 4;
```

Diese einzelnen Zustände lassen sich mit einer `switch()`-Anweisung abfragen:

```
int state = STATE_READY_TO_WATCH;
while (!gameOver) {
    switch (state) {
        case STATE_READY_TO_WATCH:
            break;

        case STATE_WATCH:
            break;
        case STATE_READY_TO_PLAY:
            break;

        case STATE_PLAY:
```

```
            break;

        case STATE_GAME_OVER:
            break;
    }
    show();
    gameOver |= key[KEY_ESC];
}
```

Zwar ist diese Methode nicht die eleganteste, aber sie funktioniert. Wir werden im Laufe dieses Buches noch eine bessere Methode erarbeiten mit Automaten umzugehen. Für den Anfang ist diese Methode aber durchaus ausreichend.

Bei einigen Zuständen wird nach einer gewissen Zeit automatisch zum nächsten gewechselt. So folgt auf den »Achtung«-Status nach einer gewissen Zeit der »Vorgabe«-Status und auf den »Bereit«-Status folgt der »Spiel«-Status.

Wir haben im Hauptprogramm einen Timer erstellt, der seinen Zähler 30 mal in der Sekunde erhöht. Um nun einen Wechsel nach einer bestimmten Zeit realisieren zu können, brauchen wir eine weitere Variable.

```
int endCounter = -1;
```

Diese Variable enthält den Zeitpunkt, an dem zu einem neuen Zustand gewechselt werden soll. Wollen wir zum Beispiel in 1.5 Sekunden einen neuen Status aufrufen, dann könnte dies so aussehen:

```
if (endCounter < 0) {
    // 45 frames = 1.5 Sekunden
    endCounter = timerCounter + 45;
} else if (timerCounter >= endCounter) {
    endCounter = -1;
    state      = STATE_WATCH;
}
```

Wenn der `endCounter` einen Wert kleiner 0 hat, dann bedeutet dies, dass erst gerade eben in diesen Zustand gewechselt wurde. In diesem Fall berechnen wie den Endzeitpunkt. Da `timerCounter` 30 mal in der Sekunde erhöht wird, entspricht ein Unterschied von 45 einer Zeit von 1.5 Sekunden.

Sobald `timerCounter` diesen Wert erreicht oder überschritten hat, ist es Zeit zum nächsten Zustand zu wechseln. Der `endCounter` wird wieder auf –1 gesetzt, damit der neue Status auch ggf. einen time out setzen kann.

Dann wird der Variablen state ein neuer Wert zugewiesen und beim nächsten Schleifendurchlauf wird dann der neue Status aktiv.

Der »Achtung«-Zustand

Die einzige Aufgabe dieses Zustands ist es, einen Hinweis für den Nutzer darzustellen, und nach einer gewissen Zeit zum nächsten Zustand zu wechseln.

```
case STATE_READY_TO_WATCH:
    if (endCounter < 0) {
        // 45 frames = 1.5 Sekunden
        endCounter = timerCounter + 45;
    } else if (timerCounter >= endCounter) {
        endCounter = -1;
        state      = STATE_WATCH;
    }
    blit(images[READY_IMG], doubleBuffer,
         0, 0, 0, 0,
         images[READY_IMG]->w,images[READY_IMG]->h);
    break;
```

Der »Wiedergabe«-Zustand

Hier wird dem Spieler die Folge vorgespielt, die er nachspielen soll. Alle im memory Vector enthaltenen Werte werden nacheinander abgespielt. Dabei wird jedes Element für 20 Frames angezeigt, mit fünf Frames Pause zwischen den einzelnen Elementen.

```
case STATE_WATCH:
    if (endCounter < 0) {
        // 25 frames
        endCounter = timerCounter + 25;
        play_sample(sample[memory[curIndex]],
                    255, 128, 1000, 0);
    } else if (timerCounter >= endCounter) {
        if (curIndex < memory.size()-1) {
            curIndex++;
            endCounter = -1;
        } else {
            endCounter = -1;
            state      = STATE_READY_TO_PLAY;
        }
    } else if (timerCounter >= endCounter-5) {
```

```
        blit(images[NORM_IMG], doubleBuffer,
             0, 0, 0, 0,
             images[NORM_IMG]->w,
             images[NORM_IMG]->h);
    } else {
        blit(images[memory[curIndex]], doubleBuffer,
             0, 0, 0, 0,
             images[memory[curIndex]]->w,
             images[memory[curIndex]]->h);
    }
    break;
```

Hat endCounter den Wert –1, dann wird der dem aktuellen Eintrag im memory Vector zugeordneten Sound abgespielt und der Zeitpunkt für das nächste Element gesetzt.

Ist dieser Zeitpunkt gekommen, wird überprüft, ob es noch weitere Elemente im Vector gibt. Wenn ja, dann wird auf das nächste Element gewechselt. Wenn nicht, wird der Zustand auf STATE_READY_TO_PLAY gesetzt, um den Spieler zu informieren, dass er gleich an der Reihe ist.

Sind es nur noch fünf Frames oder weniger bis zum nächsten Element (oder zum Zustandswechsel), wird die neutrale Variante des Spielfelds gezeigt, ansonsten die Variante, die dem aktuellen Wert von memory[curIndex] entspricht.

Wenn keiner dieser Spezialfälle eintrifft, dann wird nur das dem aktiven Element entsprechende Spielfeld angezeigt.

Der »Bereit«-Zustand

Dieser Zustand ist wie der »Achtung«-Zustand nur ein Zwischenbild. Bis auf das angezeigte Bild entspricht der Code 100% dem des »Achtung«-Zustandes.

Der Spiel-Status

Da dieser Zustand der umfangreichste ist, gehen wir ihn Schritt für Schritt durch.

```
if (endCounter < 0) {
    joypad->poll();
    if (joypad->button[Joypad::UP]) {
        curDir = UP;
```

```
    } else if (joypad->button[Joypad::DOWN]) {
        curDir = DOWN;
    } else if (joypad->button[Joypad::LEFT]) {
        curDir = LEFT;
    } else if (joypad->button[Joypad::RIGHT]) {
        curDir = RIGHT;
    } else {
        curDir = NONE;
    }
    if (curDir != lastDir && curDir != NONE) {
        endCounter = timerCounter + 15;
        play_sample(sample[curDir], 255, 128, 1000, 0);
    }
    lastDir = curDir;
} else if // …
```

Hat endCounter einen negativen Wert, wird der Zustand des virtuellen Joypads abgefragt. Nach dem Aufruf von poll() werden die einzelnen Richtungen überprüft und curDir auf den entsprechenden Wert gesetzt.

Nur wenn sich der Wert seit dem letzten Check geändert hat und eine Richtung ausgewählt wurde, wird endCounter auf einen Wert 15 Frames in die Zukunft gesetzt, und das der gedrückten Richtung entsprechende Soundfile abgespielt.

Wenn dieser Zeitpunkt erreicht ist, dann wird dieser Teil des Codes ausgeführt:

```
} else if (timerCounter >= endCounter) {
    endCounter = -1;
    if (memory[curIndex] == curDir) {
        // Richtige Taste
        curIndex++;
        score += 5;
        if (curIndex >= memory.size()) {
            // Ok, geschafft. Neue Zahl anhängen
            memory.push_back(rnd(4));

            // Und neu Abspielen
            state = STATE_READY_TO_WATCH;
            endCounter = -1;
            curIndex   =  0;
        }
    } else {
        // Fehler
        curIndex = 0;
```

```
            lastDir  = 0;
            tries--;

            endCounter = -1;
            curIndex   =  0;
            if (tries <= 0) {
                state = STATE_GAME_OVER;
            } else {
                // Und neu Abspielen
                state = STATE_READY_TO_WATCH;
            }
        }
}
```

Zuerst wird überprüft, ob die eingegebene Richtung mit dem entsprechenden Eintrag im `memory`-Vector übereinstimmt. Wenn ja, dann wird der aktuelle Index um eins erhöht und der Spieler erhält fünf Punkte gutgeschrieben.

Sollte damit das Ende der gespeicherten Werte erreicht worden sein, wird ein neuer Wert hinzugefügt und der Status auf `STATE_READY_TO_WATCH` geändert.

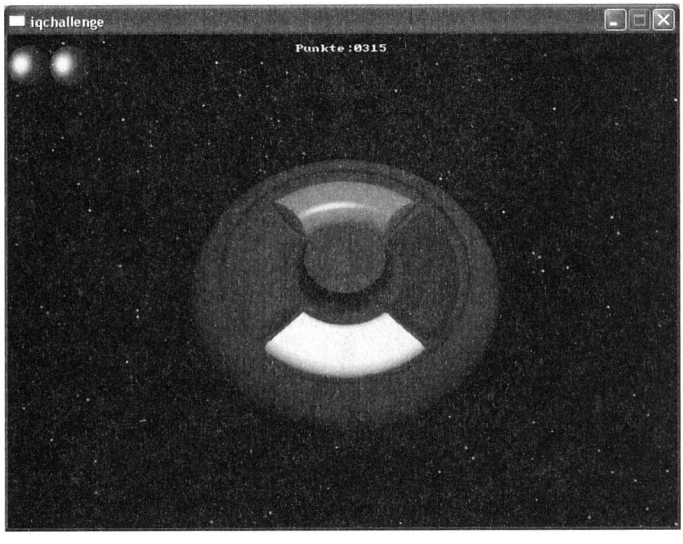


Abbildung 13.4: Screenshot des Spieles unter Windows XP

Sollten die Werte nicht übereinstimmen, wird der Index zurückgesetzt und die Anzahl der Versuche um eins verringert. Falls dies der letzte Versuch gewesen sein sollte (`tries <=0`), wird zum Spielendestatus gewechselt, ansonsten wird die Folge dem Spieler noch mal vorgespielt (`STATE_READY_TO_WATCH`).

In jedem Fall wird das aktuelle Bild angezeigt:

```
blit(images[lastDir], doubleBuffer,
     0, 0, 0, 0,
     images[lastDir]->w,images[curDir]->h);
```

Der Spiel-Ende-Status

Dieser Status dient wieder nur der Anzeige einer Nachricht (»Game Over«) an den Spieler. Der einzige Unterschied zu den anderen beiden Info-Zuständen ist, dass nach Ablauf der Zeit das Spiel-Ende-Flag gesetzt wird:

```
case STATE_GAME_OVER:
    if (endCounter < 0) {
        // 45 frames = 1.5 Sekunden
        endCounter = timerCounter + 45;
    } else if (timerCounter >= endCounter) {
        endCounter = -1;
        gameOver = TRUE;
    }
    blit(images[END_IMG], doubleBuffer,
         0, 0, 0, 0,
         images[END_IMG]->w,images[END_IMG]->h);
    break;
```

Aufräumarbeiten

Bevor die `game()`-Funktion zum Hauptprogramm zurückkehrt, wird noch der von den Bildern und Sounds belegte Speicher wieder freigegeben.

```
    for (int a=0; a < 8; a++) {
        if (a <4) {
            destroy_sample(sample[a]);
        }
        destroy_bitmap(images[a]);
    }
```

Den vollständigen Quellcode dieses Beispiels mit allen benötigten Sound- und Grafikdateien finden Sie wie immer auf der dem Buch beigelegten CD.

Erweiterungsmöglichkeiten

Bei einigen Punkten habe ich in diesem Spiel den »schnellen« Weg genommen.

- ✔ Alle Grafiken bedecken den gesamten Schirm.
- ✔ Der Schirm wird bei jedem Durchlauf der Schleife neu gemalt, unabhängig davon, ob sich etwas geändert hat oder nicht.
- ✔ Die Anzeigeschirme haben keine akustische Untermalung.

Passen Sie das Spiel doch so an, dass diese Punkte auch abgehakt werden können. Oder erweitern Sie das Spiel doch um Sprachausgabe: Schnappen Sie sich ein Mikrofon und nehmen Sie ein paar Sätze auf (»Aufgepasst!« – »Und jetzt Sie!« – »Das war wohl nichts!«) und bauen Sie die entsprechenden Soundfiles in das Spiel ein.

Viele der möglichen Erweiterungen sind kosmetischer Natur. So fehlen Menüs, ein Optionsmenü und eine High-Score-Liste. Genau mit diesen Elementen beschäftigten sich die nächsten Kapitel. Falls Sie das Spiel also abrunden möchten, dann lesen Sie die nächsten Kapitel und testen Sie Ihr neues Wissen an diesem Programm.

Spiel als Minispiel benutzen

Ein Spiel dieser Größenordnung lässt sich wunderbar als Minispiel benutzen. Die einfachste Methode wäre natürlich das Spiel wie es jetzt ist in einer Spielhalle in der Rollenspielwelt bereit zu stellen.

Es ist jedoch auch möglich dieses Spiel leicht zu verändern, damit es sich nahtlos in das Hauptspiel integriert. Benutzt man anstatt Farben Animationen der Hauptfigur, könnte man diese im Spiel tanzen lassen (als Teil eines Rätsels, um in einen Tanzclub aufgenommen zu werden oder um einen Tanzwettbewerb zu gewinnen).

Benutzt man anstatt des farbigen Spielfeldes eine Zifferntastatur, kann man damit ein Codeschloss simulieren. Der Spieler könnte beobachten, wie einer der Widersacher einen Code in dieses Schloss eingibt (natürlich sollte es ein längerer Code sein, damit es auch interessanter wird) und muss diesen dann später selbst eingeben.

14 Textausgabe

Textausgabe ist eine der grundlegenden Funktionalitäten eines Computers. Und doch haben wir uns bisher nur sehr stiefmütterlich darum gekümmert. Dies werden wir jetzt ändern. In diesem Kapitel lernen Sie Allegros Text- und Zeichensatzfunktion zu nutzen, eigene Zeichensätze zu erstellen und mit mehrfarbigen Schriften umzugehen. Zu diesem Zweck werden zwei wichtige Tools betrachtet: Der Grabber und TTF2PCX.

Funktionen zur Textausgabe

Man könnte sagen, Sie haben zwei grundsätzliche Möglichkeiten den Text auf den Bildschirm zu bringen: Entweder Sie benutzen die `textout()`-Funktionen oder die `textprintf()`-Funktionen. Der Unterschied ist, dass `textout()` nur einen String anzeigt. Bei `textprintf()` können Sie alle von `printf()` bekannten Formate nutzen, um Zahlen, Strings, einzelne Zeichen und Fließkommazahlen beliebig auszugeben.

Die textout()-Funktion

Diese grundlegende Textausgabefunktion ist wie folgt definiert:

```
void textout(BITMAP *bmp, const FONT *font,
             const char *text,
             int x, y, color);
```

Parameter	Bedeutung
`bmp`	Die Bitmap, auf der der Text erscheinen soll.
`font`	Der Zeichensatz, der zur Ausgabe benutzt werden soll.
`text`	Der Text, der ausgegeben werden soll.
`x,y`	Die Position, an der der Text ausgegeben werden soll.
`color`	Die Farbe des Textes.

Tabelle 14.1: Parameter der textout()-Funktion

Allegro definiert einen nicht-proportionalen Font (das heißt: einen Zeichensatz, in dem alle Zeichen gleich hoch und breit sind). Dieser standardmäßige Font hat den sinnigen Namen `font`. Alle Zeichen des vorgegebenen Zeichensatzes passen in ein 8x8-Pixel-Raster.

Wenn Sie den Text »Spinat macht stark!« an der Position 35, 102 in Grün auf den Schirm bringen wollen, dann könnte das so aussehen:

```
textout(screen, font, 35, 102, "Spinat macht stark!", make-
col(0,255,0);
```

Bitte stellen Sie sicher, dass Sie immer text_mode(-1) in Ihrer Initialisierungsmethode aufrufen. Wenn Sie diesen Befehl vergessen, dann wird Ihr Text immer mit einem störenden Rahmen ausgegeben.

Neben der »normalen« `textout()`-Version gibt es auch noch Varianten, die den Text zentriert oder rechtsbündig ausgeben.

Funktion	Beschreibung
`textout_centre`	Gibt den Text so aus, dass die Mitte des Textes an der angegebenen Position ist.
`textout_right`	Gibt den Text so aus, dass sich das Ende des Textes (die rechte Seite) an der angegebenen Position befindet.

Tabelle 14.2: textout_centre() und textout_right()

Alle Parameter dieser Funktionen sind identisch mit denen von `textout()`, nur die Bedeutung der `x,y` Parameter ist abhängig von der jeweiligen Version.

```
#include <allegro.h>
BITMAP   *doubleBuffer = NULL;

// Die normalen Funktionen aus der vorigen
// Kapiteln init(), done(), etc.
// Wurden aus Platzgründen aus diesem Listing
// entfernt

int main(int argc, char** argv) {
```

```
    init();
    clear(doubleBuffer);
    text_mode(-1);

    textout(doubleBuffer, font, "Links!",
            10, 10, makecol(255,0,0));
    textout_centre(doubleBuffer, font, "Mitte!",
            doubleBuffer->w/2, 50, makecol(0,255,0));
    textout_right( doubleBuffer, font, "Rechts!",
           doubleBuffer->w, 100, makecol(0,0,255));

    clear_keybuf();
    blit(doubleBuffer, screen, 0, 0, 0, 0,
         SCREEN_W, SCREEN_H);
    readkey();

    done();
} END_OF_MAIN();
```

Das Ausgeben von Zeichenketten funktioniert schon wunderbar, aber was ist, wenn man den Punktestand des Spielers ausgeben möchte?

Die textprintf()-Funktionen

Stellen Sie sich vor, Ihr Rollenspielheld kommt in die Stadt Greifenwald, nachdem er zuvor in Eisenwald die Hydra Zantila besiegt hat. Es wäre doch schön, wenn wir einen Text wie

»Willkommen $Held! Wir haben von Eurem Sieg über $Name_letztes_Monster in $Name_letzte_Stadt gehört! Auch wir haben eine Aufgabe für Euch...«

vorbereiten könnten, und dann bei der Ausgabe nur noch die Platzhalter mit den korrekten Werten ersetzen müssten. Dann wäre es vollkommen egal, ob der Spieler davor die Hydra in Eisenwald oder den Grottenschrat in Hintertupfingen besiegt hat.

Und genau für solche Zwecke gibt es die Standard-C-Funktion `printf()`. Sie nimmt einen String mit einer Formatierungsanweisung und eine Reihe von Ausgabeparametern entgegen. Allegro stellt Ihnen eine Funktion zur Verfügung, die diese Funktionalität mit `textout` verbindet: `textprintf()`. Diese Funktion ist wie folgt definiert:

```
void textprintf(BITMAP *bmp, const FONT *font,
     int x, y, color, const char *fmt, ...);
```

Auf den ersten Blick fällt auf, dass sich die Reihenfolge der Parameter geändert hat. Nun stehen die Position und Farbe vor dem Text. Dies hat den Grund, dass alle Parameter die auf den Text folgen, als Teil der Ausgabe betrachtet werden.

```
textprintf(doubleBuffer, font,
        0, 100, makecol(255,0,0),
        "Willkommen %s! Wir haben von Eurem Sieg über"
        "%s in %s gehört.",
        hero.name.c_str(), hero.lastMonster.c_str(),
        hero.lastCity.c_str());
```

Der Platzhalter `%s` steht für eine beliebige Zeichenkette. Bei der Ausgabe der Zeichenkette werden die einzelnen Platzhalter durch die übergebenen Werte ersetzt.

Es gibt eine ganze Menge von möglichen Platzhaltern:

Platzhalter	Parametertyp; Ausgabe als
d, i	`int`; Ausgabe als Dezimalzahl mit Vorzeichen
o	`int`; Ausgabe als Oktalzahl ohne Vorzeichen und führende 0
x,X	`int`; Ausgabe als Hexadezimalzahl ohne Vorzeichen und ohne ein 0x voranzustellen. Bei X werden die in der Hexadezimaldarstellung vorkommenden Zeichen groß geschrieben, bei x klein.
u	`int`; Ausgabe als Dezimalzahl ohne Vorzeichen
c	`int`; Ausgabe als Zeichen, nachdem das `int` in ein `char` gewandelt wurde.
s	`char*`; Gibt alle Zeichen bis zur ersten \0 aus, oder so viele Zeichen wie durch den Präzisionsmodifikator angegeben werden. %10s würde also maximal die ersten 10 Zeichen ausgeben.
f	`double`; Dezimale Ausgabe. Die Anzahl der Nachkommastellen hängt von der Präzison ab. Bei %2.3f würde die Zahl 1,5 so ausgegeben: »1.500«
e, E	`double`; Ausgabe in Exponenten-Schreibweise

Platzhalter	Parametertyp; Ausgabe als
g, G	`double`; Ausgabe entweder wie bei %f oder %e je nach Größe des Exponenten
p	`void*`; Ausgabe des Zeigerwertes
n	`int*`; Die Anzahl der bisher bei diesem Aufruf ausgegebenen Zeichen wird in den Parameter geschrieben. Es findet keine Ausgabe statt.
%	Kein Parameter benötigt; es wird ein % ausgegeben.

Tabelle 14.3: Formate für textprintf()

Sie können es sich so vorstellen: Wenn Sie `textprintf()` benutzen, dann erstellt Allegro für Sie einen String welcher der Formatangabe entspricht, und gibt diesen dann mit Hilfe von `textout()` aus. Wenn Sie also nur eine Zeichenkette ausgeben wollen, dann ist `textout()` immer die bessere Lösung. Wenn Sie mehrere Aufrufe von `textprintf()` mit den gleichen Formatangaben machen, dann sollten Sie den Text erst in einen String schreiben und dann diesen String mehrfach ausgeben.

Textcodierung

Wenn Sie nun den Text mit `textprintf()` ausgeben, werden Sie eine Ausgabe erhalten, die in etwa so aussieht:

```
Willkommen Gustav! Wir haben von Eurem Sieg über Zantilla in Eisen-
wald geh^rt.
```

Offensichtlich werden die Umlaute nicht korrekt dargestellt. Das Problem ist, dass der Quellcode ASCII codiert ist, Allegro jedoch per Voreinstellung von UTF-8 codiertem Text ausgeht.

Formatkonstante	Beschreibung
U_ASCII	Ein `char` pro Zeichen, 8 bit ASCII
U_ASCII_CP	Ein `char` pro Zeichen, alternative Codepage. Die Codepage kann mit `set_ucodepage()` gesetzt werden.

Formatkonstante	Beschreibung
U_UNICODE	Zwei chars pro Zeichen, 16 Bit Unicode
U_UTF8	Variable Anzahl von chars pro Zeichen, UTF-8 formatierte Unicode Zeichen

Tabelle 14.4: Allegro-Textformate

Zwar kann das Textformat theoretisch jederzeit geändert werden, allerdings sind nach dem Aufruf alle Allegro-internen Zeichenketten im falschen Format. Aus diesem Grund sollte das Textformat direkt vor dem Aufruf von `allegro_init()` gesetzt werden. Die `init()` Funktion sollte dann also in etwa so aussehen:

```
void init() {
    srand(time(NULL));

    // Allegro und Grafik, Timer, Sound
    // und Eingabemedien initialisieren
    set_uformat(U_ASCII);
    allegro_init();
    // ...
    // Grafik Modus setzten, etc.
    // ...
    // Den Rahmen um den Text verschwinden lassen
    text_mode(-1);
}
```

Sobald Sie den Aufruf von `set_uformat(U_ASCII)` in den Code eingefügt haben, werden auch die Umlaute korrekt dargestellt.

Andere Fonts benutzen

Der normale Allegro-Font ist zwar OK, wenn man nur kurz den Wert einiger Variablen zu Debug-Zwecken ausgeben möchte, aber für das eigentlich Spiel ist er dann doch entschieden zu nüchtern.

Ein kurzer Blick in die Allegro-Dokumentation offenbart aber keine Funktion, um einen neuen Zeichensatz zu laden. Was nun? Diese Frage wird sehr häufig gestellt und die Antwort ist auch nicht offensichtlich. Der Weg zum eigenen Zeichensatz führt über zwei Programme:

- ✔ TTF2PCX – Ein Programm, um aus Windows Fonts eine PCX Datei zu machen.
- ✔ Grabber – Ein Allegro Programm, welches eigentlich dazu gedacht ist, alle externen Dateien (Grafiken/Sounds etc) zu einer einzigen DAT-Datei zusammenzufassen.

TTF2PCX

Dieses Programm erlaubt es Ihnen aus normalen Fonts eine PCX Datei zu erstellen.

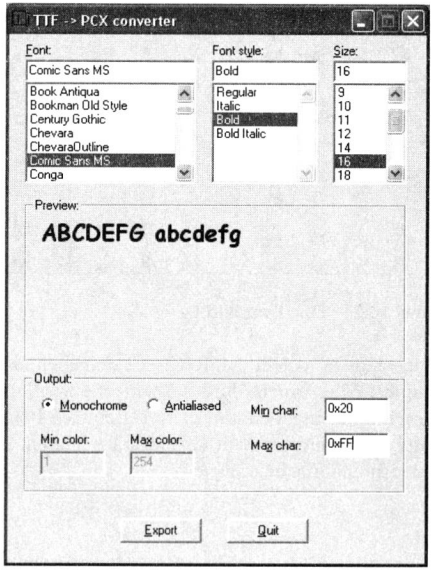


Abbildung 14.1: TTF2PCX

Sie finden TTF2PCX auf der Buch CD

Stellen Sie sicher, dass Sie im Feld *Max Char* `0xFF` eingeben, damit auch die Umlaute enthalten sind. Das noch nicht ganz so eindrucksvolle Resultat sieht in etwa so aus:

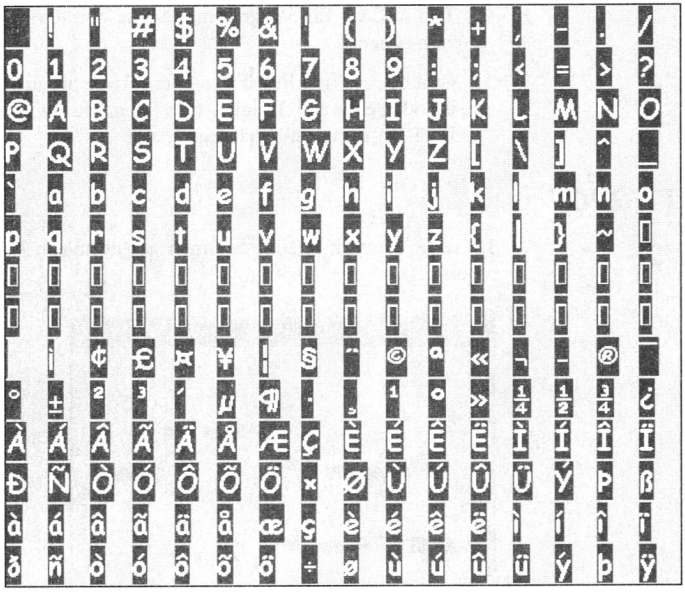


Abbildung 14.2: Eine Font Bild-Datei

Jeder Buchstabe ist von einem Rechteck umgeben, das die Größe des Zeichens angibt. Alles innerhalb der Rechtecke wird als Zeichen aufgefasst. Sie könnten nun hingehen und diese Datei nach Belieben verändern oder eine Datei in diesem Stil von Grund auf erstellen. Aber mit TTF2PCX geht es deutlich einfacher.

Grabber

Um nun aus dem PCX einen Font zu machen gehen Sie wie folgt vor: Im *allegro/tools*-Verzeichnis finden Sie den Grabber. Starten Sie das Programm, und lassen Sie sich von der doch etwas ungewöhnlich anmutenden Optik nicht beunruhigen.

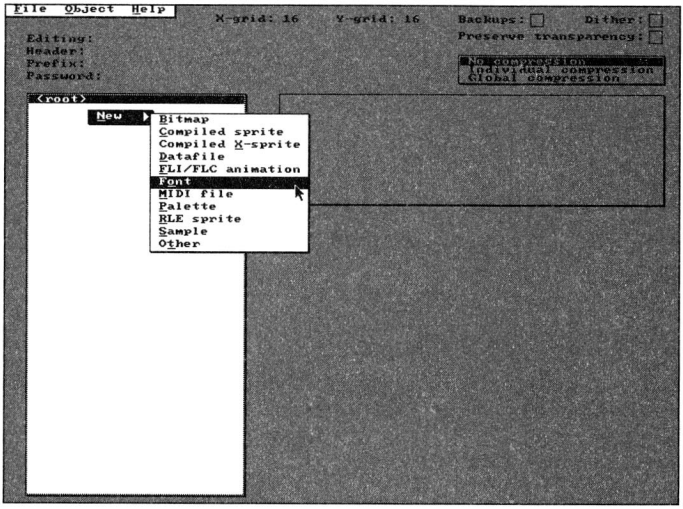


Abbildung 14.3: Der Grabber

Nach einem Rechtsklick auf einen leeren Bereich in der weißen Fläche wählen Sie *New / Font*. Daraufhin öffnet sich eine Dialogbox, die nach dem Namen des Objekts fragt. Bestätigen Sie mit OK, ohne einen Namen einzugeben (siehe Abbildung 14.4).

Nun noch ein Rechtsklick auf den `FONT` Eintrag in der Liste. Wählen Sie *Grab* im Popup-Menü. Im nun aufgehenden *Datei öffnen*-Dialog wählen Sie bitte die gerade erstellte PCX-Datei aus. Lassen Sie sich von dem für Windowsverhältnisse sehr ungewöhnlich aussehenden Dialog nicht beirren (siehe Abbildung 14.5).

Wenn alles funktioniert hat, sollten Sie nun in der rechten unteren Ecke eine Voransicht des Fonts sehen können (siehe Abbildung 14.6).

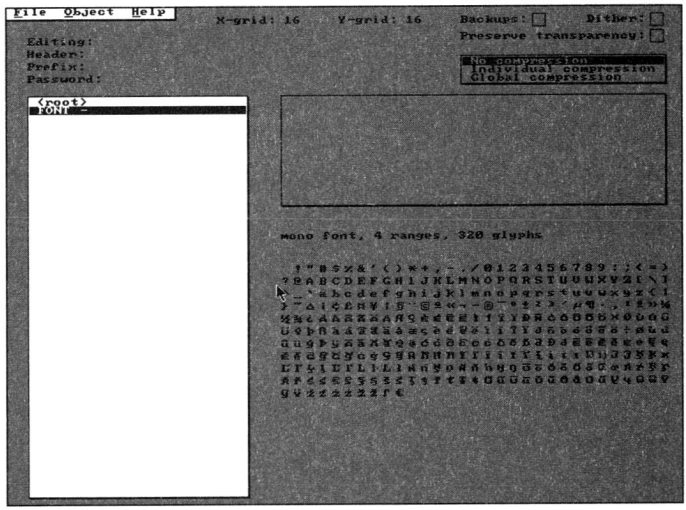


Abbildung 14.4: Ein erstellter Font

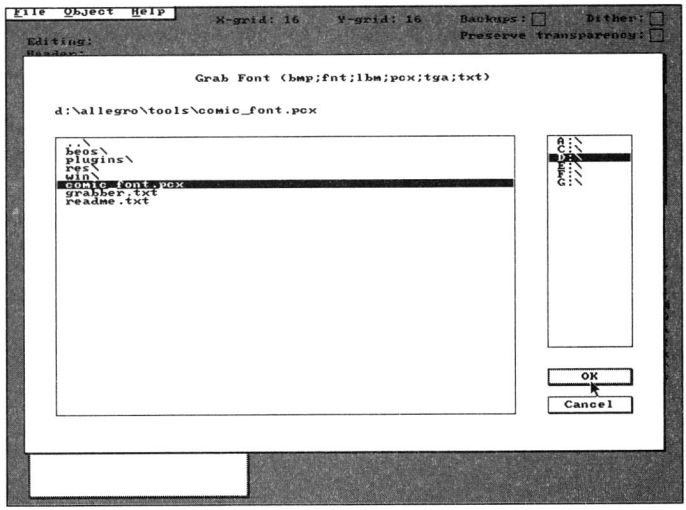


Abbildung 14.5: Der *Datei öffnen*-Dialog

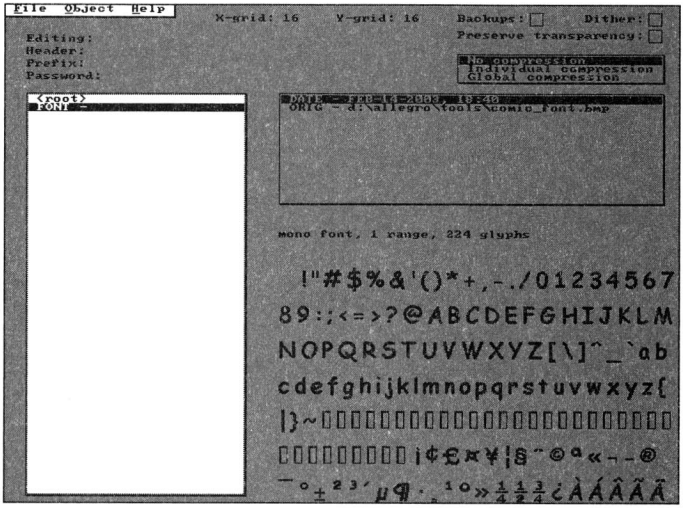


Abbildung 14.6: Der endgültige Font

Bevor Sie den Zeichensatz speichern, sollten Sie noch in der Auswahlbox am rechten oberen Rand »Global compression« auswählen. Nun speichern Sie die Datei mit *File / Save* oder [Strg] + [S]. Ein passender Name wäre zum Beispiel der Name des Fonts, gefolgt von der Größe und der Endung ».dat«. Also zum Beispiel »*comic16.dat*«.

Jetzt müssen wir den Font nur noch im Programm laden.

Einen eigenen Font laden

Wir haben jetzt ein Dat-File mit einem Font. Um diese Datei zu laden, benutzen wir `load_datafile()`

```
DATAFILE *load_datafile(const char *filename);
```

Der einzige Parameter dieser Funktion ist der Name der Datei, die geladen werden soll. Der Rückgabewert ist ein Array mit den jeweiligen Einträgen des Dat Files.

```
DATAFILE *dat;
FONT     *comicFont;
dat       = load_datafile("comic.dat");
comicFont = (FONT*) dat[0].dat;
```

Bis jetzt hat das Dat File nur einen Eintrag. Also können wir gefahrlos davon ausgehen, dass der Zeichensatz an der ersten Stelle steht. Nach der Zuweisung von `dat[0].dat` an `comicFont` haben wir einen gültigen Font im Speicher den wir überall da verwenden können, wo wir bisher den Default-Zeichensatz von Allegro benutzt haben.

Zeichensätze in Farbe

Eine Besonderheit der auf Grafiken basierenden Allegro-Zeichensätze ist, dass sie vollkommen farbig sein können. Es gibt nur ein paar wenige Einschränkungen:

- ✔ Das Bild muss mit 8-Bit-Farbtiefe (256 Farben) gespeichert werden.
- ✔ Farbindex 0 muss »magisches Pink« sein. RGB 255, 0, 255
- ✔ Farbindex 255 muss Gelb sein. RGB 255, 255, 0
- ✔ Das Pink darf nur für die Rechtecke um die Zeichen benutzt werden.
- ✔ Das Gelb darf nur für den Hintergrund benutzt werden

Ansonsten können Sie sich total austoben. Ein gold-metallic Font könnte in etwa so wie in Abbildung 14.7 aussehen.

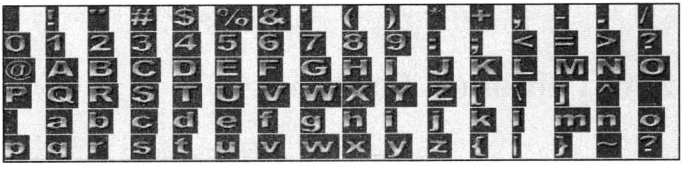


Abbildung 14.7: Ein goldener Font

Bei einem farbigen Font muss neben dem eigentlichen Zeichensatz auch die Palette gespeichert werden. Dies erfordert einen weiteren Schritt im Grabber.

Nachdem Sie den Font erzeugt und von der Bilddatei »ge-grabbt« haben, erzeugen Sie noch ein Palettenobjekt, indem Sie nach einem Rechtsklick auf die Liste *New/Palette* auswählen. Klicken Sie mit der rechten Maustaste auf das neue Objekt und wählen Sie *Grab*. Wählen Sie die Bitmap aus, von der Sie den Zeichensatz importiert haben. Sie sollten nun die Palette mit allen Farben, die im Zeichensatz verwendet sehen können.

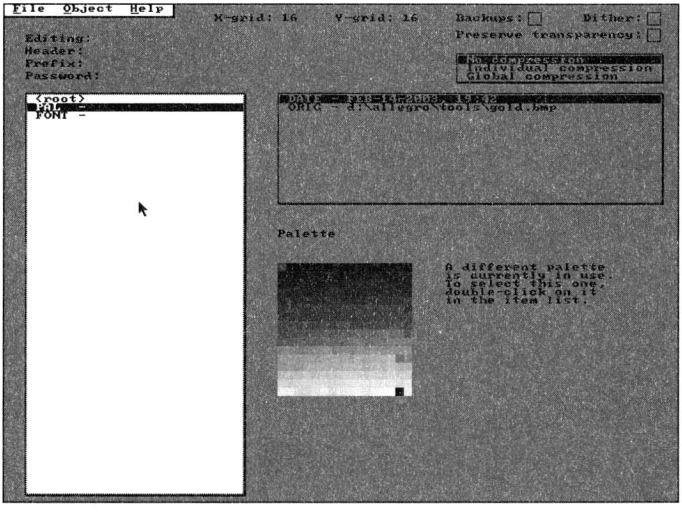


Abbildung 14.8: Eine Palette in Grabber

Fügen Sie die Palette immer nach dem Font hinzu. Dadurch wird sicher gestellt, dass die Palette auch nach dem Zeichensatz gespeichert wird.

Der Rest bleibt wie gehabt. Setzen Sie die Kompression auf »*Global compression*« und speichern Sie dann das File mit einem sprechenden Namen ab.

Das Laden des Dat-Files bleibt wie gehabt. Sie müssen jedoch vor dem Anzeigen des Textes noch die Palette setzen.

```
DATAFILE *goldDat;
FONT     *goldFont;
```

```
goldDat  = load_datafile("gold.dat");
goldFont = (FONT*) goldDat[0].dat;
set_palette((RGB*) goldDat[1].dat);
```

Wenn Sie jetzt der Meinung sind, dass dieses Vorgehen doch eher umständlich ist, dann muss ich Ihnen Recht geben. Für die nächste »große« Allegro-Version ist auch geplant, Zeichensätze auf eine komfortablere Weise zu laden. Bis dahin ist jedoch ein Dat-File die einzige Möglichkeit.

Natürlich finden Sie alle zugehörigen Quellcodes auf der Buch CD!

15 Menüs und Dialoge

Das Erste, was der Spieler sieht, wenn er Ihr Spiel startet, ist das Hauptmenü. Schon an diesem Punkt bildet er sich eine erste Meinung über das Spiel – und in Anbetracht der Unmenge an Spielen, die es heutzutage frei erhältlich im Internet gibt, sollte der Ersteindruck so gut wie möglich sein. Wenn der Spieler Probleme hat sich im Spiel zurechtzufinden, sind die Chancen, dass er das Spiel für längere Zeit begutachtet, eher gering.

In diesem Kapitel geht es um die Erstellung von Menüs und Dialogen für Spiele. Vom Hauptmenü über Optionen bis hin zu Schaltflächen im Spiel. Die Benutzeroberfläche ist einer der wichtigsten Punkte im Spiel.

Gestaltung von Menüs und Dialogen

Ich hatte mir vor einiger Zeit eine DVD eines Films ausgeliehen. Nach dem Einlegen präsentierte sich mir ein erstes Menü, in dem ich zwischen der deutschen und englischen Sprachversion des Filmes wählen sollte. Der Schirm sah in etwa so aus wie in dieser Abbildung:

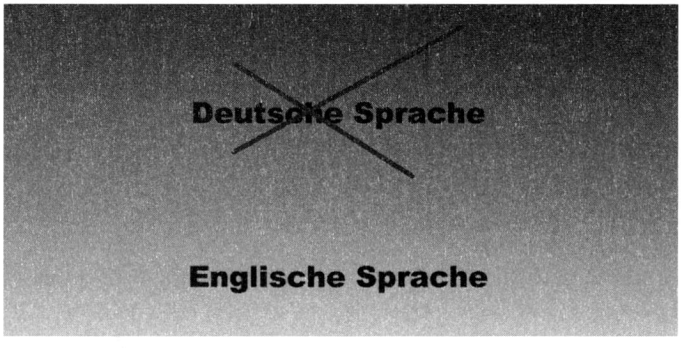


Abbildung 15.1: Nachstellung eines suboptimalen DVD-Menüs

Mein Problem war einfach: Welcher der beiden Einträge war nun aktiv? Der angekreuzte? Oder war der Eintrag durchgestrichen? Mit den Cursortasten hoch und runter zu schalten brachte mich auch nicht viel wei-

ter, da ja nur zwei Einträge da waren. Also habe ich geraten und mich natürlich prompt für den falschen Eintrag entschieden.

Die Moral von der Geschichte ist: In einem Menü sollte immer klar zu erkennen sein, welcher Eintrag aktiv ist.

Farben

Wenn Sie sich dafür entscheiden, dies durch eine Änderung der Farbe zu bewerkstelligen, haben Sie ein Problem, wenn sich in dem Menü nur zwei Einträge befinden. Sobald Sie mehr als zwei Einträge in einem Menü haben, ist es wieder offensichtlich, welcher angewählt ist. Jedenfalls wenn ihre Farben einen guten Kontrast haben und der Spieler nicht farbenblind ist. Ansonsten kann es passieren, dass einige Spieler nicht in der Lage sind, überhaupt einen Unterschied festzustellen.

Größe

Neben der Farbe können Sie auch die Größe ändern. Größere Schrift wird meist als wichtiger und im Vordergrund stehend betrachtet und ist deswegen leicht als ausgewählter Eintrag zu erkennen.

Abbildung 15.2: Aktives Element durch Größe hervorgehoben

Marker

Eine weitere Möglichkeit ist es, den aktuellen Eintrag durch einen Marker zu kennzeichnen. Sie können das aktive Element einrahmen oder eine optische Markierung auf eine oder beide Seiten des Eintrages stellen.

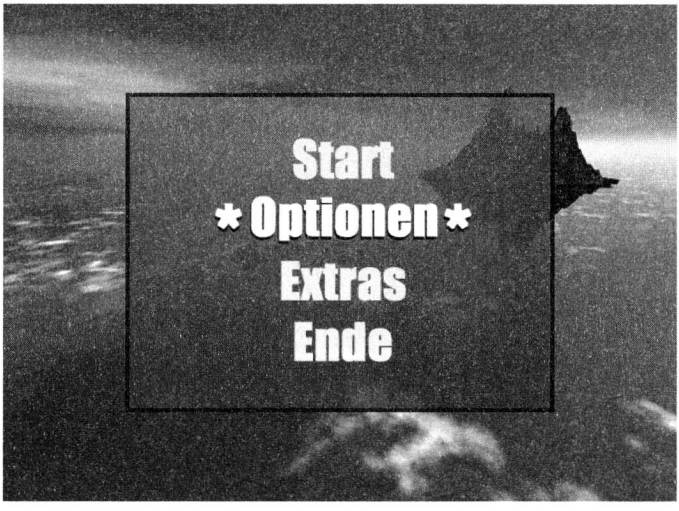


Abbildung 15.3: Aktives Element durch Markierung hervorheben

Animation

Ein weiteres sehr effektives Mittel ist es, das selektierte Element zu animieren. Der Menüeintrag könnte auf- und abhüpfen, die Größe könnte sich pulsierend ändern, die Buchstaben könnten etwas rotieren etc.

Abbildung 15.4: Aktives Element ist animiert

Sprachausgabe

Eine weitere Möglichkeit ist es, das selektierte Element über Sprachausgabe bekannt zu geben. Lassen Sie doch einfach eine Stimme »Spielstart« oder »Optionen« sagen. Wenn Sie dies mit einer der anderen genannten Methoden kombinieren, ist die Wahrscheinlichkeit, dass jemand mit Ihrem Menü Probleme hat sehr gering.

Mausunterstützung

Wenn Ihr Spiel mausbasiert ist, dann wird das Element selektiert, sobald die Maus im Bereich des Elements ist. Der Mauszeiger funktioniert wie ein vom Spieler kontrollierter Marker. Wichtig ist, dass die gesamte Benutzeroberfläche einheitlich ist. Wenn Sie in Ihren Menüs die Maus unterstützen, dann müssen Sie die Maus auch im eigentlichen Spiel unterstützen. Ist das eigentliche Spiel mausbasiert, dann müssen auch die Menüs komplett mit der Maus bedienbar sein. Mischformen, in denen das Spiel auf die eine Weise kontrolliert wird (zum Beispiel: nur mit der Maus oder nur mit dem Joystick) dann ist es wichtig, dass der Spieler

nicht das Eingabegerät wechseln muss, nur um einen Menüpunkt zu bestätigen.

Implementierung eines Menüs

Menüs werden wohl in beinahe allen Spielen gebraucht werden. Aus diesem Grund wird der komplette Code, den wir in diesem Kapitel erarbeiten, in eine eigene Quellcode Datei ausgelagert, damit wir ihn möglichst einfach weiterverwenden können.

Menülisten

Menülisten sind die Art von Menüs, die man in der Regel auf dem Titelbildschirm sieht: Eine Liste von anwählbaren Punkten, die untereinander stehen. Diese Art von Menüs ist sehr einfach zu realisieren. Wir benötigen eine Liste mit Einträgen und eine Variable, welche die aktuelle Position speichert. Beim Anzeigen der Einträge vergleichen wir die aktuelle Position mit der selektierten Position und zeigen den Eintrag dementsprechend an. Um wirklich sicher zu gehen, dass der Spieler erkennt, welcher Eintrag selektiert ist, wird für das aktive Element sowohl eine andere Farbe als auch ein anderer Font benutzt.

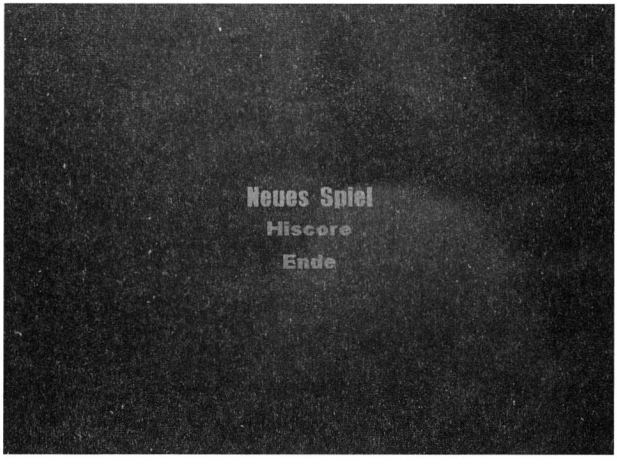


Abbildung 15.5: Die Menü-Routine in Aktion

Die einfachste Lösung für ein solches Menü sieht in etwa so aus:

```
// Simples Menü –
// Mit einigen Verbesserungsmöglichkeiten
#include <allegro.h>
#include "util.h"

FONT *menuNormal   = NULL;
FONT *menuSelected = NULL;

int mainMenu() {
    char* items[] = {
        "Neues Spiel",
        "Hiscore",
        "Ende",
        (char*)NULL,
    };

    int minX  = SCREEN_W;
    int maxX  = 0;
    int maxL  = 0;
    int count = 0;

    while (items[count] != NULL) {
        int len = text_length(menuNormal, items[count]);
        maxL    = MAX(len, maxL);
        len     = text_length(menuSelected,
                              items[count]);
        maxL    = MAX(len, maxL);
        count++;
    }
    minX = (SCREEN_W - maxL)/2;
    maxX = (SCREEN_W + maxL)/2;

    int dist   = text_height(menuSelected)+2;
    int startY = (SCREEN_H - (count * dist))/2;

    int selected     = 0;
    int lastSelected = -1;
    BITMAP *bg = load_bitmap("bg.tga", NULL);

    int mode = joypad->getPollMode();
    joypad->setPollMode(Joypad::REPORT_STATE_CHANGE);

    int done = FALSE;
```

```
while (!done) {
    blit(bg, doubleBuffer,0,0,0,0,bg->w,bg->h);

    joypad->poll();
    if (joypad->button[Joypad::DOWN]) {
        ++selected;
        if (selected >= count) {
            selected = 0;
        }
    } else if (joypad->button[Joypad::UP]) {
        --selected;
        if (selected < 0) {
            selected = count -1;
        }
    } else if (joypad->button[Joypad::ACTION]) {
        done = TRUE;
    } else if (joypad->button[Joypad::MENU]) {
        done = TRUE;
        selected = -1;
    }

    for (int a=0; a < count; a++) {
        if (a == selected) {
            textout_centre(
                doubleBuffer, menuSelected,
                items[a]    , SCREEN_W/2,
                startY + dist * a + 1,
                makecol(255, 128, 0)
            );
        } else {
            textout_centre(
                doubleBuffer, menuNormal,
                items[a]    , SCREEN_W/2,
                startY + dist * a + ABS(dist -
                 text_height(menuNormal))/2,
                makecol(128, 128, 128)
            );
        }
    }

    if (selected != lastSelected) {
        lastSelected = selected;
        blit(doubleBuffer, screen, 0, 0, 0, 0, SCREEN_W,
       SCREEN_H);
    }
}
```

```
    joypad->setPollMode(mode);
    destroy_bitmap(bg);
    return selected;
}

int main(int argc, char** argv) {

    init(640,480,60);

    DATAFILE *fonts = load_datafile("fonts.dat");
    menuNormal   = (FONT*) fonts[0].dat;
    menuSelected = (FONT*) fonts[1].dat;

    mainMenu();

    unload_datafile(fonts);

    return 0;
} END_OF_MAIN();
```

Dieser Codeabschnitt ist einem tatsächlichen Spiel entnommen. Allerdings sollten Ihrem geschulten Auge einige Dinge auffallen, die nicht so glücklich gelöst sind. Falls Sie den Code überblättert haben, dann ist jetzt ein guter Zeitpunkt, ihn etwas genauer anzusehen und nach Schwachstellen zu suchen. Und es gibt einige Probleme in diesem Code.

Fangen wir oben an: Die beiden Fonts werden global deklariert, aber nur innerhalb der Funktion benutzt. Wobei man in Spielen nicht immer etwas gegen globale Variablen sagen kann, sind sie in diesem Fall recht schmerzlos zu vermeiden.

Ein paar Zeilen tiefer ist der zweite Hammer: Das Menü bekommt keinerlei Parameter und definiert die Texte innerhalb der Funktion. Dies macht es nicht gerade einfach, den Code wiederzuverwenden. Würde man das Array mit den Strings als Parameter übergeben (sagen wir mal: gemeinsam mit den Fonts), dann könnte man diese Funktion für verschiedene Menüs benutzen.

Kurz darunter geht es weiter: Der Dateiname des Hintergrundbildes ist fest in die Funktion codiert. Spätestens an diesem Punkt sollte Sie ein leichtes Gruseln überkommen.

Der verbleibende Code ist in Ordnung – allerdings sieht dieses Menü es nicht vor, dass ein Eintrag auch »ausgegraut«, also nicht anwählbar sein kann.

Die einzige Art und Weise diesen Code wiederzuverwenden besteht darin, ihn mittels Copy'n'Paste zu übernehmen und dann anzupassen. Oder Sie benutzen ihn als Ausgangsbasis für etwas Flexibleres.

Welche Eigenschaften muss eine flexible Menüroutine haben?

- ✔ Sie muss wiederverwendbar sein.
- ✔ Alle Aspekte müssen weitestgehend anpassbar sein.
  - ✔ Hintergrund (Farbe/Bild)
  - ✔ Darstellung der Einträge, besonders der Selektion
- ✔ Einfach zu benutzen
- ✔ Unterstützung für normale, selektierte und ausgegraute Elemente

Die Art der Darstellung kann stark variieren – wir haben ja weiter oben bereits einige Möglichkeiten besprochen. Um für ausreichend Flexibilität zu sorgen, kann man die Grundfunktionen in einer Basisklasse realisieren und die konkreten Umsetzungen dann in den abgeleiteten Klassen.

```
using namespace std;

class AbstractMenu {
public:
    enum {
        ENABLED     = 0,
        DISABLED    = 1,
        SELECTED    = 2,

        COUNT_MODES = 3
    };

    AbstractMenu() : selectedIndex(0), model() {
    }
    virtual ~AbstractMenu() {
        for (StringVectorIterator iter = model.begin(); iter <
        model.end(); ++iter) {
            delete *iter;
        }
    }

    virtual void drawBackground(BITMAP *dest) = 0;
    virtual void drawItem(BITMAP *dest,
                          const char* text,
```

```
                        int x, int y,
                        int w, int h,
                        int mode) = 0;
virtual int runMenu(BITMAP *dest) = 0;

virtual int moveSelection(int delta) {
    int next  = selectedIndex ;
    int count = model.size();
    int tries = 0;

    do {
        next += delta;
        if (next < 0) {
            next += count;
        } else if (next >= (int) model.size()) {
            next -= count;
        }
        tries++;
        // Kein selektierbares Element vorhanden
        if (tries >= count) {
            return -1;
        }
    } while (!isEnabled(next));
    return next;
}

virtual void addElement(string *data) {
    model.push_back(new string(*data));
}
virtual void addElement(char *data) {
    model.push_back(new string(data));
}
virtual string* getElement(int index) {
    return model[index];
}

virtual void setEnabled(int index, int enable) {
    while (index >= (int) enabled.size()) {
        enabled.push_back(TRUE);
    }
    enabled[index] = enable;
}

virtual int isEnabled(int index) {
    if (index < (int)enabled.size()) {
        return enabled[index];
```

```
        }
        return TRUE;
    }

protected:
    int selectedIndex;
    StringVector model;
    vector<int>  enabled;
};
```

Diese Klasse definiert nur die grundlegenden Aktionen eines Menüs. Sie können Elemente hinzufügen, Sie können Elemente auf inaktiv setzten (»ausgrauen«) und Sie können den selektierten Index mit einer beliebigen Schrittweite ändern.

Die Texte werden in einem Vector von `std::string`-Zeigern gespeichert:

```
typedef std::vector<std::string*> StringVector;
typedef StringVector::iterator   StringVectorIterator;
```

Beim Einfügen des Strings legen wir eine Kopie dieser Zeichenkette an. Dadurch ist der gesamte Inhalt des Vectors in unserer Kontrolle – es steht ebenfalls fest, dass wir diese Strings wieder freigeben müssen. Aus Bequemlichkeitsgründen stellen wir auch eine Version von `addElement()` zur Verfügung, die eine C-Zeichenkette als Parameter bekommt.

Des Weiteren haben wir einen Vector, der speichert, ob ein bestimmtes Element anwählbar ist oder nicht. Wenn Sie keines der Elemente ausgrauen wollen, dann wird der Vector auch nicht mit Daten gefüllt. Erst wenn ein Element auf inaktiv gesetzt wird, dann wird es notwendig, den Zustand aller Elemente bis zum inaktiv gesetzten parat zu haben:

```
    virtual void setEnabled(int index, int enable) {
        while (index >= (int) enabled.size()) {
            enabled.push_back(TRUE);
        }
        enabled[index] = enable;
    }
```

Der Grund dafür ist einfach: Die `isEnabled()`-Methode überprüft erst, ob der angefragte Index in dem Vector gespeichert ist. Ist er das nicht, dann wird `TRUE` zurückgeliefert. Ansonsten der Wert des angefragten Element-Indices.

Das Kernstück dieser Klasse ist die `runMenu()`-Methode – diese übernimmt das eigentliche Anzeigen des Menüs. Oder sagen wir: Sie würde es tun, wenn sie implementiert wäre.

Die runMenu()-Methode ist jedoch eine rein virtuelle Methode – sie wird erst in abgeleiteten Klassen implementiert. Genauso verhält es sich mit den beiden Methoden zur Darstellung des Menüs.

drawBackground übernimmt die komplette Darstellung des Hintergrundes. Sie müssen diese Methode überladen, um einen Hintergrund für Ihr Menü anzuzeigen.

Die drawItem-Methode übernimmt die Anzeige der einzelnen Einträge. Sie bekommt ein Rechteck, in dem die Elemente angezeigt werden sollen und den Zustand (ENABLED, DISABLED, SELECTED) übergeben. Was sie damit macht, ist komplett ihr überlassen.

Das AbstractMenu hat schon jetzt die Größe der ersten hart codierten Funktion erreicht, und das obwohl sie noch kein einziges Zeichen auf den Schirm bringt. Sobald Sie aber mehr als ein Menü in Ihrem Spiel haben oder das Menü anpassen wollen, macht sich dieser extra Aufwand mehr als bezahlt.

Und sobald wir die fehlenden Funktionen in einer abgeleiteten Klasse implementiert haben, ist das Menü auch schon wieder einsatzfähig.

Diese neue Klasse soll sich prinzipiell so verhalten wie die Ausgangsfunktion, also ein Hintergrundbild anzeigen, die Einträge mit verschiedenen Fonts darstellen können etc.

Die eher einfachen Methoden dieser Klasse dienen also nur dazu diese Attribute zu verwalten.

```
class SimpleMenu : public AbstractMenu {

public:
    SimpleMenu() : bg(NULL), selected(0) {
        normal   = NULL;
        selected = NULL;
        for (int a=0; a < COUNT_MODES; a++) {
            color[a] = 0;
        }
    }

    virtual void drawBackground(BITMAP *dest) {
        if (bg) {
            blit(bg, dest, 0, 0, 0, 0, bg->w, bg->h);
        } else {
            clear(dest);
        }
    }
```

```
    void setBackground(BITMAP *img){
        bg = img;
    }
    void setColor(int mode, int color) {
        if (mode >= 0 && mode < COUNT_MODES) {
            this->color[mode] = color;
        }
    }
    void setFonts(FONT* norm, FONT* sel) {
        normal    = norm;
        selected  = sel;
    }
private:
    BITMAP *bg;
    FONT   *normal, *selected;
    int    color[COUNT_MODES];
};
```

Ich sagte ja, dass es sich hierbei um die einfachen Methoden handelt.

Diese Methoden erlauben es Ihnen, Hintergrund, Farben und Zeichensätze nach Ihrem Belieben zu setzen. Allerdings haben wir bisher die beiden Arbeitspferde der neuen Klasse außen vor gelassen.

```
    virtual void drawItem(BITMAP *dest,
                          const char* text,
                          int x, int y,
                          int w, int h, int mode) {

        FONT *fnt = NULL;
        if (mode != SELECTED) {
            fnt = (normal != NULL) ? normal : font;
        } else {
            fnt = (selected != NULL) ? selected : font;
        }
        x += w/2;
        y += (h- text_height(fnt))/2;
        textout_centre(dest, fnt, text,
                       x, y, color[mode]);
    }
```

Die `drawItem`-Methode macht es sich auch sehr einfach. Sie wählt den Font je nach übergebenem Modus, berechnet dann den Mittelpunkt des Bereichs in dem sie sich darstellen soll, und ruft am Ende `textout_centre` auf, um die eigentliche Arbeit zu machen.

Und nun kommt das Kernstück dieser Klasse: die `runMenu()`-Methode.

```
virtual int runMenu(BITMAP *dest) {
        int done    = FALSE;
        int joyMode = joypad->getPollMode();
        int last    = -1;
        int y       = 0;
        int curY    = 0;
        int delta   = 0;

        delta = normal
                  ? text_height(normal)
                  : text_height(font);
        delta = MAX(delta, normal
                       ? text_height(selected)
                       : text_height(font));

        y = (SCREEN_H - (delta * model.size()))/2;

        joypad->setPollMode(
                       Joypad::REPORT_STATE_CHANGE);
        while (!done) {
            joypad->poll();
            if (joypad->button[Joypad::UP]) {
                selectedIndex = moveSelection(-1);
            } else if (joypad->button[Joypad::DOWN]) {
                selectedIndex = moveSelection(+1);
            } else if (joypad->button[Joypad::ACTION]){
                done = TRUE;
            } else if (joypad->button[Joypad::MENU]) {
                done = TRUE;
                selectedIndex = -1;
            }

            if (selectedIndex != last) {
                last = selectedIndex;
                drawBackground(dest);

                curY = y;
                for (int a=0;a<(int)model.size();a++){
                    int mode = DISABLED;
                    if (isEnabled(a)) {
                        if (a==selectedIndex) {
                            mode = SELECTED;
                        } else {
```

```
                        mode = ENABLED;
                    }
                }
                drawItem(dest, model[a]->c_str(),
                         0, curY, SCREEN_W, delta,
                         mode);
                curY+=delta;
            }
            blit(dest, screen, 0, 0, 0, 0,
                 dest->w, dest->h);
        }
    }
    joypad->setPollMode(joyMode);
    return selectedIndex;
}
```

So viel Neues gibt es hier gar nicht zu sehen. Diese Funktion ist in erster Linie die Schleife aus der ersten Menüfunktion, nur werden diesmal Klassen-Methoden aufgerufen werden, damit die Darstellung flexibler ist. Und natürlich werden auch die ausgegrauten Elemente korrekt behandelt.

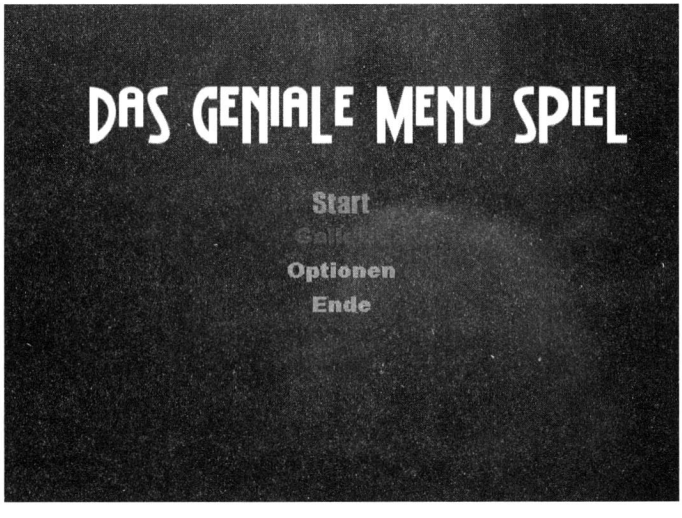


Abbildung 15.6: Die Menü-Klasse in Aktion

Bei einem Menü handelt es sich noch um ein einfaches Oberflächenelement. Wenn es jedoch an komplexere Dinge geht, dann übersteigt der Aufwand, das Interface zu programmieren in manchen Fällen die Zeit, die für das Erstellen des eigentliche Spiel benötigt wird.

Allegros GUI-Routinen

Allegro bietet einen recht umfangreichen Satz an Bedienelementen. Es gibt Listen, Buttons, Textausgabeelemente und einiges mehr. All diese Elemente haben eines gemeinsam: Sie können sowohl mit der Maus und Tastatur als auch mit dem Joystick bedient werden. Des Weiteren sehen all diese Elemente normalerweise nicht sehr ansprechend aus.

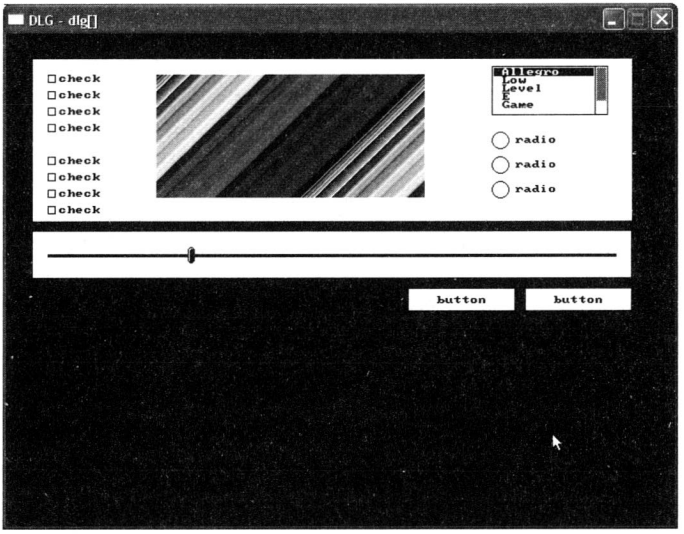


Abbildung 15.7: Einige Allegro GUI-Elemente

Glücklicherweise kann man das Aussehen der Elemente selbst beeinflussen. Und genau das werden wir tun. Bevor wir uns aber in die tiefen Abgründe der Allegro GUI-Routinen werfen, möchte ich Ihnen noch ein sehr nützliches Tool vorstellen: den Allegro Dialog Editor.

DLG – a Dialog Editor

Mit Hilfe dieses von Julien Cugniere geschriebenen Tools können Sie auf äußert leichte Art und Weise Dialoge erstellen. Nach Auswahl der gewünschten GUI-Routinen können Sie sie einfach auf dem Bildschirm positionieren, gruppieren und bewegen.

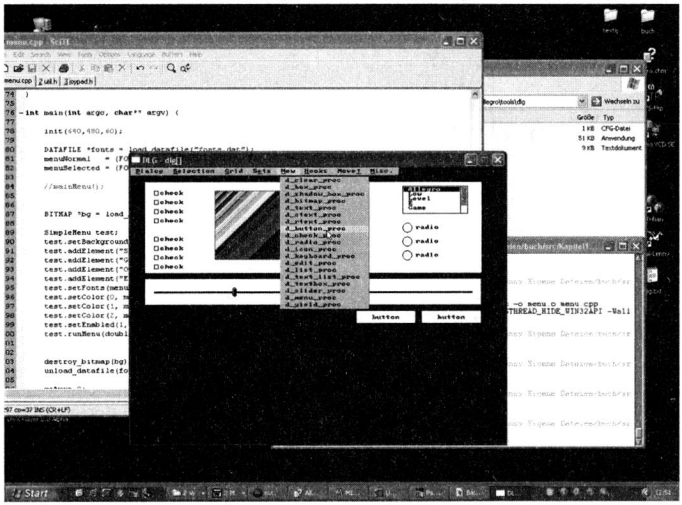


Abbildung 15.8: Dialoge mit DLG erstellen

Ich schlage vor, Sie öffnen das Programm, und erstellen ein paar Bedienelemente. Wählen Sie dazu einfach aus dem Menü eine Routine aus, und ziehen Sie dann im Arbeitsbereich ein Rechteck mit der Maus auf. In diesem Bereich erscheint dann das Element. Sie können jedes Element nach dem Erstellen bewegen und vergrößern wie es Ihnen beliebt. Wenn Sie sich nicht sicher sind, was die einzelnen Methoden machen, dann blättern Sie einfach etwas vor und spicken Sie im nächsten Abschnitt.

DLG finden Sie auf der beigelegten CD, sowohl als Quellcode als auch in einer kompilierten Version für Windows.

Aufbau der GUI-Routinen

Obwohl Allegro eine C-Bibliothek ist, benutzen die GUI-Routinen eine Art von Objektorientierung. Zwar ist diese nicht so ausgefeilt wie die in C++ eingebaute, dennoch lassen sich die einzelnen Bedienelemente sehr einfach anpassen.

Ein Dialog wird durch ein Array von `DIALOG`-Strukturen gebildet. Der letzte Eintrag in diesem Array muss NULL sein.

Attribut	Verwendung
`int (*proc)(int, DIALOG *, int)`	Die Funktion, die die Nachrichten empfängt und auf sie reagiert. Diese Funktion definiert um was für eine Art von Objekt es sich handelt.
`int x, y, w, h`	Position und Größe des Objekts
`int fg, bg`	Vorder- und Hintergrundfarbe
`int key`	ASCII-Wert für ein mögliches Tastenkürzel
`int flags`	Flags, die den Status des Objekts angeben.
`int d1, d2`	Benutzerdefiniert oder abhängig von der Dialogfunktion.
`void *dp, *dp2, *dp3`	Benutzer oder Dialogsfunktion abhängiger Zeiger auf beliebige Daten

Tabelle 15.1: Die DIALOG-Struktur

Der wichtigste Eintrag ist hier die `Dialog`-Prozedur. Sie hat immer den Aufbau:

```
int prozedur_name(int msg, DIALOG *d, int c);
```

und bestimmt das Verhalten des GUI-Elements. Eigentlich ist diese Funktion das GUI-Element. Der erste Parameter, `msg`, ist die Nachricht, die an die Funktion übergegeben wird. Dies könnte der Befehl sein, sich selbst zu zeichnen, die Benachrichtigung über einen Tastendruck oder einfach auch nur ein Hinweis, dass sich die Maus nun über diesem Element befindet.

Dieses DIALOG-Array ist aus dem leicht abgewandelten exgui.c-Beispiel. Die einzigen Änderungen bestehen darin, die Erweiterung der Datei auf ».cpp« zu ändern und einige Casts einzufügen um den Compiler von der Richtigkeit der übergebenen Parameter zu überzeugen. Des Weiteren wurde die Beschriftung abgeändert. Normalerweise steht jeder DIALOG-Eintrag in einer Zeile – allerdings wäre diese Darstellung bei der geringen Breite dieses Buches nicht lesbar gewesen.

```
#define VP void*
#define END_OF_DIALOG {NULL,0,0,0,0,0,0,0,0,0,NULL,\
                       NULL,NULL}
DIALOG the_dialog[] ={
    { d_clear_proc,            // (dialog proc)
      0,    0,    0,    0,     // (x)   (y)   (w)   (h)
      255,  0,                 // (fg)  (bg)
      0,                       // (key)
      0,                       // (flags)
      0,                       // (d1)
      0,                       // (d2)
      (VP) NULL,               // (dp)
      NULL,                    // (dp2)
      NULL                     // (dp3)
    },
    { d_edit_proc,             // (dialog proc)
      80,   32,   512,  48,    // (x)   (y)   (w)   (h)
      255,  0,                 // (fg)  (bg)
      0,                       // (key)
      0,                       // (flags)
      sizeof(the_string)-1,    // (d1)
      0,                       // (d2)
      (VP)the_string,          // (dp)
      NULL,                    // (dp2)
      NULL                     // (dp3)
    },
    { d_button_proc,           // (dialog proc)
      80,   132,  161,  49,    // (x)   (y)   (w)   (h)
      255,  0,                 // (fg)  (bg)
      't',                     // (key)
      0,                       // (flags)
      0,                       // (d1)
      0,                       // (d2)
      (VP)"&Toggle Me",        // (dp)
      NULL,                    // (dp2)
      NULL                     // (dp3)
    },
```

```
    { d_list_proc,            // (dialog proc)
      360,  100,  207,  207,  // (x)   (y)   (w)   (h)
      255,  0,                // (fg)  (bg)
      0,                      // (key)
      0,                      // (flags)
      0,                      // (d1)
      0,                      // (d2)
      (VP)listbox_getter,     // (dp)
      NULL,                   // (dp2)
      NULL                    // (dp3)
    },
    { change_font_proc,       // (dialog proc)
      80,   232,  161,  49,   // (x)   (y)   (w)   (h)
      255,  0,                // (fg)  (bg)
      'f',                    // (key)
      D_EXIT,                 // (flags)
      0,                      // (d1)
      0,                      // (d2)
      (VP)"Change &Font",     // (dp)
      NULL,                   // (dp2)
      NULL                    // (dp3)
    },
    // … Jede Menge Code der auch nicht anders aussah
    END_OF_DIALOG
};
```

Wenn Sie dieser Code erschreckt hat, dann kann ich Sie beruhigen: Sie werden solchen Code nie selbst erstellen müssen. Diese Aufgabe können Sie beruhigt DLG überlassen.

Um diesen Dialog anzuzeigen, benutzen Sie ein Stück Code, das in etwa so aussieht:

```
ret = do_dialog(the_dialog, -1);
```

Diese Funktion zeigt den Dialog auf dem Schirm an, kümmert sich um alle Arten von Benutzereingaben und beendet den Dialog erst dann, wenn eines der Elemente eine `D_EXIT`-Nachricht verschickt (normalerweise eine `d_button_proc` – also eine normale Schaltfläche).

Einige Dialog-Prozeduren im Überblick

Es würde deutlich den Rahmen dieses Buches sprengen, auf alle Dialogfunktionen einzugehen. Darüber hinaus wäre dies auch nicht gerade aufregend. Aus diesem Grund werde ich mich darauf beschränken Ihnen zu

zeigen, wie man mit Hilfe der Allegro GUI-Routinen (Böse Zungen würden sagen: trotz der Allegro GUI-Routinen) eine ansprechende Oberfläche für ein Spiel erstellen kann.

Es gibt ein paar grundlegende Dinge, die Sie wissen sollten: Zuerst einmal: die Allegro GUI-Routinen zeichnen immer direkt auf den `screen`. Wenn Sie die Elemente auf einer anderen Bitmap darstellen wollen, dann besteht Ihre einzige Chance darin, den Wert von `screen` kurzfristig zu ändern. Um dies zu tun, müssen Sie das Verhalten von `do_dialog()` ändern. Glücklicherweise ist Allegro Open Source. Ein Blick in die entsprechende Datei zeigt uns, dass `do_dialog()` wie folgt implementiert ist:

```
/* do_dialog:
 *  The basic dialog manager. The list of dialog
 *  objects should be terminated by one with a null
 *  dialog procedure.Returns the index of the object
 *  which caused it to exit.
 */
int do_dialog(DIALOG *dialog, int focus_obj)
{
   BITMAP *mouse_screen = _mouse_screen;
   int screen_count = _gfx_mode_set_count;
   void *player;

   if (!is_same_bitmap(_mouse_screen, screen))
      show_mouse(screen);

   player = init_dialog(dialog, focus_obj);

   while (update_dialog(player))
      ;

   if (_gfx_mode_set_count == screen_count)
      show_mouse(mouse_screen);

   return shutdown_dialog(player);
}
```

Die Routine speichert erst einige interne Daten, zeigt dann die Maus auf dem `screen` an, initialisiert den Player (Der Player speichert die Dialog-Informationen zwischen den Aufrufen von `update_dialog()`.) und ruft dann solange `update_dialog()` auf, bis der Dialog beendet wird. Dann wird der Dialog heruntergefahren und der Index der Dialogprozedur zurückgegeben, die die `D_EXIT`-Nachricht gesendet hat.

Mit diesem Wissen können wir nun die Funktionalität von do_dialog() entweder nachbilden oder in unsere Hauptschleife integrieren.

```
DIALOG_PLAYER *player       = NULL;
player = init_dialog(the_dialog, -1);

int   done         = FALSE;
int   needsRefresh = TRUE;
timerCounter = 0;
while (!done) {
    if (timerCounter) {
        BITMAP *orgScreen = screen;
        screen = doubleBuffer;
        while (timerCounter) {
            if (!update_dialog(player)) {
                done = TRUE;
            }
            timerCounter--;
        }
        screen = orgScreen;
        needsRefresh = TRUE;
    }
    if (needsRefresh) {
        needsRefresh = FALSE;
        show();
    }
    done += key[KEY_ESC];
}
```

Auf diese Weise kann die GUI in der normalen Hauptschleife und mit einem Double Buffer benutzt werden.

Schönere Buttons

Sobald sich ein GUI-Element malen soll, ruft die update_dialog()-Funktion die entsprechende Dialogprozedur auf und übergibt ihr den Befehl MSG_DRAW: »Zeichne Dich Selbst«.

Wenn wir also wollen, dass der Button (oder ein anderes Element) besser aussieht, dann müssen wir nur auf diese Nachricht reagieren.

```
int prettyButton(int msg, DIALOG*d, int c) {
    int result = D_O_K;
    if (msg == MSG_DRAW) {
        // Unsere eigene Zeichen Routine
```

```
    } else {
        // Die normale Button Prozedur soll sich
        // um den Rest kümmern.
        result = d_button_proc(msg, d, c);
    }
    return result;
}
```

Jetzt kommt die eigentliche Frage: Wie soll denn unsere Schaltfläche aussehen? Wir können entweder die Zeichenfunktionen von Allegro benutzen und den neuen Buttons mit Hilfe von Kreisen, Rechtecken und ähnlichen Figuren zusammenstellen, oder wir benutzen die Bitmap-Funktionen und erstellen aus diesen die Schaltflächendarstellung.

Beide Methoden haben Vor- und Nachteile. Für die Bitmapvariante spricht, dass sie in der Regel einfach besser aussieht. Allerdings braucht man für jeden Zustand (normal, gerückt, inaktiv etc.) eine eigene Bitmap.

Zeichnet man den Button von Hand, dann reicht es meistens die Farben auszutauschen oder einfach nur eine Zeichenaktion wegzulassen. Der Aufwand ist also meist deutlich geringer, das Ergebnis meist aber nicht so ansprechend wie bei der Bitmap-Lösung.

Ich werde hier jetzt auf das Zeichnen von Hand eingehen. Im nächsten Kapitel werden wir dann Bitmaps benutzen um den gleichen Effekt zu realisieren.

Zustände eines GUI-Elements

Jedes GUI-Element hat einen internen Status, aus dem hervorgeht, wie es sich zeichnen muss und wie es zu reagieren hat. Dieser Zustand wird im `flags`-Attribut der `DIALOG`-Struktur gespeichert. In dieser Variable können die folgenden Bits gesetzt sein.

Bit Flag	Bedeutung
`D_EXIT`	Dieses Element schickt eine »Dialog beenden«-Nachricht sobald es aktiviert wird.
`D_SELECTED`	Dieses Objekt ist selektiert (zum Beispiel eine gedrückte Schaltfläche).
`D_GOTFOCUS`	Dieses Element hat den Tastaturfokus.

Bit Flag	Bedeutung
D_GOTMOUSE	Die Maus befindet sich derzeit im Bereich des Elements.
D_HIDDEN	Das Element ist versteckt und sollte nicht angezeigt werden.
D_DISABLED	Das Element ist inaktiv (ausgegraut) und darf nicht auf Aktionen des Benutzers reagieren.
D_DIRTY	Das Objekt muss neu gezeichnet werden.
D_INTERNAL	Nur für die interne Verwendung
D_USER	Alle Bits, die auf D_USER folgen, sind frei zur Verwendung.

Tabelle 15.2: Zustände von GUI Elementen

Für das Darstellen des Buttons sind neben dem normalen Zustand die Zustände D_SELECTED und D_DISABLED am wichtigsten.

```
if (msg == MSG_DRAW) {
    if (d->flags & D_SELECTED) {
        // Pressed
    } else if (d->flags & D_DISABLED) {
        // Inaktiv
    } else {
        // Normal
    }
} else {
    result = d_button_proc(msg, d, c);
}
```

Wenn wir eine Schaltfläche im 3D-Look mit abgerundeten Ecken zeichnen wollen, dann können wir für alle drei Zustände den gleichen Zeichencode nehmen und brauchen nur die Farben anzupassen.

In Abbildung 15.9 können Sie den 3D-Button mit abgerundeten Ecken in seinen drei Zuständen sehen. In der rechten unteren Ecke ist ein farblich angepasster, normaler Allegro-Button.

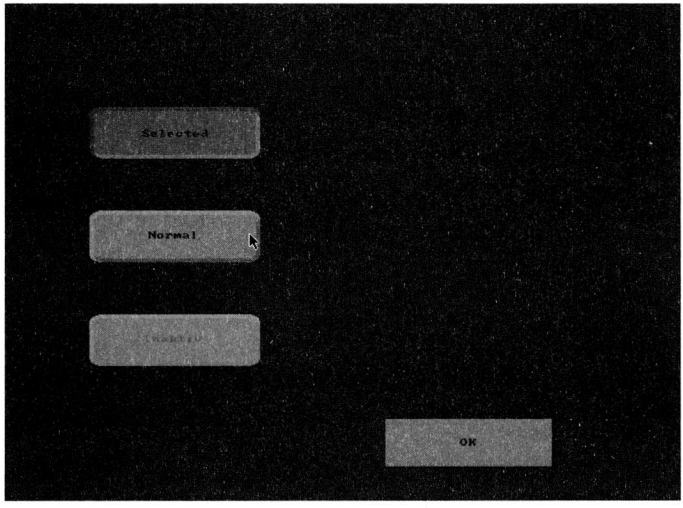


Abbildung 15.9: Zustände und Aussehen des Buttons

Bleibt nur noch ein Problem: Wie zeichnet man ein Rechteck mit abgerundeten Ecken? Allegro bietet nur Funktionen zum Zeichnen von normalen Rechtecken. Allerdings gibt es auch Funktionen zum Zeichnen von Kreisen. Da wir ein ausgefülltes Rechteck wollen, brauchen wir uns nicht mit dem Zeichnen von Kreisbögen zu beschäftigen, sondern können gefüllte Kreise in die Ecken des Rechtecks setzen, und dann gefüllte Rechtecke in den verbleibenden Bereichen zeichnen.

```
void drawRoundRect(BITMAP *dest, int x, int y,
                   int w, int h,
                   int rx, int ry, int color) {

    // Ellipsen an den Eckpunkten
    ellipsefill(dest, x + rx, y + ry     , rx, ry,
                                            color);
    ellipsefill(dest, x + rx, y + h - ry, rx, ry,
                                            color);
    ellipsefill(dest, x + w - rx, y + ry    , rx, ry,
                                            color);
    ellipsefill(dest, x + w - rx, y + h - ry, rx, ry,
```

```
                                                    color);

    // und der Hauptbereich
    rectfill(dest, x+rx  , y   , x + w - rx, y + h,
                                                    color);
    rectfill(dest, x     , y+ry, x+rx      , y + h- ry,
                                                     color);
    rectfill(dest, x+w-rx, y+ry, x+w       , y + h- ry,
                                                     color);
}
```

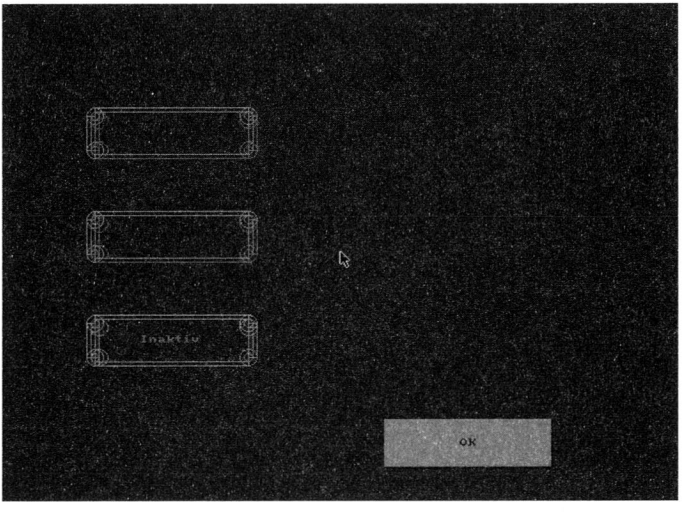


Abbildung 15.10: »Drahtgitter« der Buttons

Damit wäre die erste Hürde genommen. Allerdings stellt sich noch die Frage, mit welchen Farben die 3D-Kanten des Buttons gemalt werden. Die `DIALOG`-Struktur definiert nur zwei Farben, `fg` und `bg`. Wenn wir mehr Farben brauchen, können wir entweder die verbleibenden Felder in der `DIALOG`-Struktur benutzen (`d1`, `d2` und `d3`) oder wir berechnen die Farben anhand der vorhandenen. Wir brauchen Funktionen, die aus einer gegebenen Farbe eine hellere und dunklere Version berechnen können – und eine, die eine Farbe in Graustufen umrechnet (für die inaktive Version).

Um eine Farbe heller zu machen, muss man den Rot-, Grün- und Blauanteil der Farbe gleichmäßig erhöhen, muss aber darauf achten, dass kein Wert die 255 überschreiten darf.

```
int getLighterColor(int col) {
    int r = MIN((int) (getr(col) * 1.2), 255);
    int g = MIN((int) (getg(col) * 1.2), 255);
    int b = MIN((int) (getb(col) * 1.2), 255);

    return makecol(r,g,b);
}
```

Die resultierende Farbe ist in etwa ein Fünftel heller als die Ausgangsfarbe (wenn man von möglichen Rundungsfehlern und Werten >255 absieht).

Auf die gleiche Weise kann man auch eine dunklere Farbe berechnen. Nur muss man mit einem Wert kleiner als 1.0 multiplizieren:

```
int getDarkerColor(int col) {
    int r = MIN((int) (getr(col) * 0.8), 255);
    int g = MIN((int) (getg(col) * 0.8), 255);
    int b = MIN((int) (getb(col) * 0.8), 255);

    return makecol(r,g,b);
}
```

Um die Helligkeit und damit den Grauton einer Farbe bestimmen zu können, muss man wissen, dass die verschiedenen Farbkomponenten unterschiedlich stark zur Helligkeit beitragen.

So ist Grün (RGB 0,255, 0) in etwa 6 mal heller als Blau (RGB 0,0,255) und doppelt so hell wie Rot (RGB 255, 0, 0). Aus diesem Grund muss man die einzelnen Farbkomponenten gewichtet in den Graustufenwert einrechnen.

```
int getGrayValue(int col) {
    float r = getr(col) * 0.3f;
    float g = getg(col) * 0.59f;
    float b = getb(col) * 0.11f;

    float l = r+g+b;
    return makecol((int) l, (int) l, (int) l);
}
```

Nachdem wir nun den Grundaufbau der Dialog-Prozedur und alle Hilfsmethoden beschrieben haben, können wir nun (endlich) den Button codieren.

```
int prettyButton(int msg, DIALOG* d, int c) {
    int result = D_O_K;
    int offset = 0;

    if (msg == MSG_DRAW) {
        int shade[3];
        int fg = d->fg;
        if (d->flags & D_SELECTED) {
            // Pressed
            shade[0] = d->bg;
            shade[1] = getDarkerColor(d->bg);
            shade[2] = getDarkerColor(getDarkerColor(d->bg));
            offset+=2;
        } else if (d->flags & D_DISABLED) {
            // Nicht aktiv
            shade[0] = getGrayValue(d->bg);
            shade[1] = getGrayValue(d->bg);
            shade[2] = getGrayValue(getLighterColor(d->bg));

            fg = getDarkerColor(shade[1]);
        } else {
            shade[0] = getDarkerColor(d->bg);
            shade[1] = d->bg;
            shade[2] = getLighterColor(d->bg);
        }

        drawRoundRect(screen, d->x, d->y, d->w, d->h, 8,8, shade[2]);
        drawRoundRect(screen, d->x+4, d->y+4, d->w-4, d->h-4, 8,8,
       shade[0]);
        drawRoundRect(screen, d->x+4, d->y+4, d->w-8, d->h-8, 4,4,
       shade[1]);


        gui_textout(screen, (char*) d->dp, d->x + d->w/2+offset, d->y
        + offset+(d->h-text_height(font))/2, fg, TRUE);
    } else {
        result = d_button_proc(msg, d, c);
    }
    return result;
}
```

In Abhängigkeit vom Zustand werden die Farbwerte des Buttons berechnet. Ist die Schaltfläche selektiert, wird auch der Text leicht nach unten rechts verschoben dargestellt, um den Eindruck des »Gedrückt seins« zu verstärken.

Und das war es auch schon. Auf diese Weise können Sie nicht nur die Buttons, sondern auch jedes andere GUI-Element, das Allegro Ihnen bietet, anpassen. Natürlich können Sie auch komplett neue Elemente entwerfen. Es liegt bei Ihnen.

16 Das Quizspiel

In diesem Kapitel werden wir ein komplettes Quizspiel entwerfen und umsetzen. Dies erlaubt es Ihnen, das im letzten Kapitel angeeignete Wissen an einem tatsächlichen Spiel zu erproben.

Wirft man einen Blick in das aktuelle Fernsehprogramm, dann sieht man auf anhieb, dass Quizspiele derzeit sehr beliebt sind. Im Internet stehen auch haufenweise freie Versionen der beliebtesten Fernsehshows (natürlich mit abgewandelten Namen, um Markenrechtsproblemen aus dem Weg zu gehen) zum Download bereit.

Neben dem Aspekt, dass diese Art von Spielen gerade sehr beliebt ist, gibt es noch einige weitere Gründe sich mit ihnen zu beschäftigen:

- ✔ Sie lassen sich gut als Minispiele verwenden.
- ✔ Es ist immer spaßig sein Wissen zu testen, vor allem bei Nischengebieten wie Science Fiction und Fantasy (oder auch noch spezieller: bestimmte Science-Fiction-Serien wie Star Trek, Akte X, Herr der Ringe etc.).
- ✔ Es ist ein sehr gutes Beispiel für den Einsatz der GUI-Funktionen.

Wenn Sie sich entschließen ein Quizspiel über einen sehr spezialisierten Bereich zu machen (z.B. ein »Captain-Future«-Quiz), dann sollten Sie sich auf jeden Fall Gedanken über die Urheberrechtsfrage machen. Allerdings sollte es keine Probleme geben, wenn Sie nur Fragen über den Bereich stellen und keine Grafiken und Musikstücke benutzen, die urheberrechtlich geschützt sind. Auch das Markenrecht spielt hier eine Rolle. Suchen Sie sich einen Titel aus, der nicht zu nahe an einem geschützten Titel liegt. »Wer wird Quiz-Millionär?« könnte Ihnen ebenso wie »Das Pokemon-Quiz« Probleme bereiten.

Wenn Sie aber unbedingt einen geschützten Namen oder auch Bilder/Musik aus dem jeweiligen Themenbereich nutzen wollen, dann fragen Sie den Rechteinhaber unbedingt um Erlaubnis.

Planung

Vor der Umsetzung des Spieles müssen erst noch Umfang, Format und Art des Quiz geklärt werden.

- ✔ Wie viele Fragen werden dem Spieler gestellt?
- ✔ Gibt es Hilfen (Joker)?
- ✔ Sind die Fragen in Wissensgebiete unterteilt?
- ✔ Wird das Wissensgebiet angezeigt?
- ✔ Können zusätzlich zur Frage auch Grafiken angezeigt werden?
- ✔ Können zusätzlich zur Frage Sounds abgespielt werden?
- ✔ Wie viele Antwortmöglichkeiten hat der Spieler?
- ✔ Wie viele Fragen darf er falsch beantworten?
- ✔ Wann ist das Quiz beendet?
- ✔ Gibt es ein Zeitlimit?
- ✔ Steigt der Schwierigkeitsgrad?
  - ✔ Weniger Zeit zur Lösung?
  - ✔ Schwierigere Fragen?
  - ✔ Mehr Antworten?
- ✔ Kann der Spieler aus verschiedenen Wissensgebieten wählen?

Und nun wollen wir diese Fragen beantworten.

Spielablauf

Dem Spieler werden zehn Fragen gestellt. Er bekommt vier mögliche Antworten präsentiert. Er hat für jede Antwort zehn Sekunden Zeit. Ist die Antwort korrekt, dann bekommt er die verbleibende Zeit auf sein Punktekonto gutgeschrieben.

Zusätzlich zu einer Frage kann auch ein Bild angezeigt und eine Wave-Datei abgespielt werden.

Ist der Spieler sich nicht sicher, kann er einen Joker einsetzen. Jeder der drei Joker hat den Effekt, dass das Zeitlimit auf 30 Sekunden angehoben wird, und dass bei korrekter Antwort maximal fünf Punkte gutgeschrieben werden.

- ✔ »50:50-Joker«: Zwei falsche Antworten fallen weg.

- ✔ »Richtig?-Joker« Der Spieler kann prüfen lassen, ob eine bestimmte Antwort richtig oder falsch ist.
- ✔ »Tausch-Joker« Die Frage wird durch eine andere ersetzt.

Nach der zehnten Frage kann der Spieler sich in eine Bestenliste eintragen, falls seine Punktzahl ausreichend ist.

Die Fragen kommen aus verschiedenen Themengebieten. Allerdings wird das Themengebiet nicht angezeigt, sondern dient nur dazu, eine bessere Verteilung der Fragen zu gewährleisten. Es wird also nach Möglichkeit immer versucht, eine Frage aus einem Bereich zu wählen, der schon länger nicht an der Reihe war.

Benutzerführung

Das Spiel ist mausgesteuert. Bis auf die Eingabe seines Namens beim Eintrag in die Bestenliste muss der Spieler die Tastatur nicht benutzen.

Die einzelnen Screens

Hauptmenü

Das Hauptmenü hat vier Einträge:

- ✔ Neues Spiel
- ✔ Bestenliste
- ✔ Information
- ✔ Ende

Diese werden über einem Hintergrundbild angezeigt.

»Neues Spiel« verzweigt zum Quizscreen. »Bestenliste« zeigt die High-Score-Liste an. »Information« gibt Infos über das Spiel und »Ende« beendet das Spiel.

Die Bestenliste

Zeigt die zehn besten Spieler mit ihrer Punktezahl an. Am unteren Ende des Schirms gibt es einen »Zurück«-Button, der auf das Hauptmenü verzweigt.

Information

Zeigt eine Info-Bitmap an. Alle Information sind direkt auf dieser Bitmap enthalten, es wird kein zusätzlicher Text angezeigt. Am unteren Ende befindet sich ein »Zurück«-Button, durch den der Spieler in das Hauptmenü zurückkehren kann.

Quiz-Screen

Der Quizbildschirm ist grob in zwei Bereiche unterteilt: Im unteren Teil befinden sich die Frage und die vier Antwortmöglichkeiten. Die drei Schaltflächen für die Joker befinden sich genau am unteren Rand.

Der obere Bereich ist für die Darstellung des mit der Frage verbundenen Bildes reserviert. Ist der Frage kein Bild zugeordnet, dann ist dieser Bereich frei und man kann das Hintergrundbild sehen.

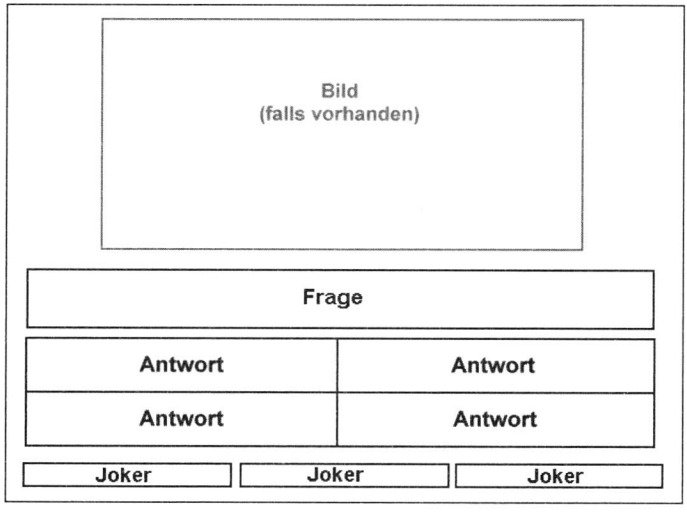


Abbildung 16.1: Aufteilung des Quizscreens

Bei Aktivierung eines Jokers wird erst eine Bestätigungsabfrage eingeblendet (»Wollen Sie den [Joker] wirklich benutzen?«). Wird diese bejaht,

tritt der Joker in Kraft. Beim »Richtig?«-Joker kann der Spieler dann noch die Frage wählen, die er überprüfen will.

Eintrag in die Bestenliste

Hat der Spieler eine Platzierung unter den ersten zehn erreicht, kann er seinen Namen eingeben. Die Eingabe des Namens erfolgt über ein normales Allegro GUI-Feld, bestätigt wird durch einen Klick auf die OK-Schaltfläche.

Soundeffekte

Die folgenden Ereignisse werden mit Sounds untermalt:

- ✔ Klicken eines Menüpunkts, Buttons oder einer Antwort
- ✔ Bestätigung bei richtiger Antwort (Jubel)
- ✔ Hinweis auf falsche Antwort (Hupe)

Mögliche Erweiterungen: Ein virtueller Moderator könnte bestimmte Sätze sprechen (Sprachausgabe). Die folgenden Sätze bieten sich an:

- ✔ »Sie meinen also, dass der [Joker] Ihnen hier weiterhilft?«
- ✔ »Das ist korrekt!«
- ✔ »Leider falsch.«

Um zu verhindern, dass diese Sätze schnell eintönig werden, sollte man allerdings für diese Phrasen mehrere Varianten bereithalten. Also nicht nur: »Das ist korrekt!«, sondern auch »Richtig!«, »Da liegen Sie goldrichtig!«, »An dieser Antwort gibt es nichts auszusetzen!« und so weiter.

Sollte dieser Weg einer starken verbalen Komponente gegangen werden, dann würde es sich anbieten, jede Frage auch gesprochen zu hinterlegen. Dies bringt aber einen deutlichen Mehraufwand mit sich.

Datenhaltung

Zu jeder Frage müssen die folgenden Informationen gespeichert werden:

- ✔ Wortlaut der Frage
- ✔ Vier Antworten

- ✔ Den Index der richtigen Antwort
- ✔ Kategorie der Frage
- ✔ Möglicherweise der Dateiname eines Bildes
- ✔ Möglicherweise der Dateiname eines Soundfiles

Man könnte sich den Index der richtigen Antwort sparen, indem man die korrekte Antwort immer auf Index 1 anführt, und dann bei der Anzeige die Antworten in eine zufällige Reihenfolge bringt. Allerdings erlaubt es dies dann nicht, über die Reihenfolge der Antworten ein Wortspiel zu starten.

Die Allegro-Config-Routinen eignen sich für diese Art von Daten nur bedingt, da sie einen eindeutigen Namen für jedes Element verlangen. Wir müssten also entweder für jede Frage eine eigene Sektion anlegen und dann die Sektionen durchnummerieren oder die Fragen und Antworten direkt durchnummerieren. Allerdings erschwert eine fortlaufende Nummerierung das Pflegen der Daten.

Stellen Sie sich vor, dass nicht nur Sie, sondern auch andere Leute Fragen für dieses Spiel entwerfen. Um eine fortlaufende Nummerierung benutzen zu können, muss jeder Fragenautor genau die vorgegebene Anzahl von Fragen verfassen – und beim Schreiben bereits den vorgegebenen Zahlenbereich verwenden.

Einfacher wäre es, wenn die Fragen einfach nur nacheinander in einer Textdatei stehen würden. In diesem Fall würde sich die Nummerierung durch die Position in der Datei ergeben, und neue Fragen können einfach durch Copy & Paste hinzugefügt werden.

Eine solche Textdatei könnte in etwa wie folgt aussehen:

```
Frage: Welche Registriernummer hat die USS-Enterprise?
Kategorie: 1
Antwort1: USS 4711
Antwort2: NCC 1701
Antwort3: HSS 2000
Antwort4: ISBN 6122
Korrekt: 2

Frage: Wer malte dieses Bild?
Kategorie: 2
Bild: mona.bmp
Antwort1: Leonardo DaVinic
```

```
Antwort2: Leonardo DiCaprio
Antwort3: Van Gough
Antwort4: Van Halen
Korrekt: 1
```

Implementierung

Dieses Spiel lässt sich hervorragend mit den Allegro-GUI-Routinen lösen. Wir werden also erst die Dialoge für die einzelnen Screens entwerfen (mit dem DLG-Tool), und dann den verbleibenden Code, um die Screen zusammenzufügen – und auch, um die Allegro Dialoge etwas hübscher zu machen.

Der Menü-Screen

Öffnen Sie DLG und stellen Sie die Auflösung auf 640 x 480 bei entweder 15-oder 16-Bit-Farbtiefe (`Misc/Graphic mode`). Erstellen Sie nun eine `d_bitmap_proc` über die ganze Fläche. Das geht am einfachsten, wenn Sie erst die Menüleiste mit einem Klick auf `Move!` verschieben, und dann den Bereich aufziehen. Nach einem Doppelklick in die erstellte `d_bitmap_proc` sollten Sie in etwa das in Abbildung 16.2 dargestellte Bild haben.

Als Nächstes erzeugen Sie 4 `d_button_procs` möglichst in der Mitte des Schirms. Legen Sie die Schaltflächen von oben nach unten an. Sie können sich etwas Zeit sparen, wenn Sie nur den ersten Button über das Menü erstellen und dann diesen mittels Strg + C und Strg + V kopieren.

Und das war es auch schon. Lassen Sie sich von dem etwas rauen Charme des Dialogs nicht abschrecken. Wenn wir im eigentlichen Programm das Hintergrundbild einfügen und schönere Buttons verwenden sieht das gleich ganz anders aus (siehe Abbildung 16.3).

Speichern Sie diesen Dialog in einer Datei namens »`menu.cpp`« und nennen Sie den Dialog »`menuDlg`«. Stellen Sie sicher, dass die »Position Dialog at (0,0)«-Checkbox ein Häkchen hat. Dann klicken Sie OK und das Grundgerüst für das Menü ist fertig.

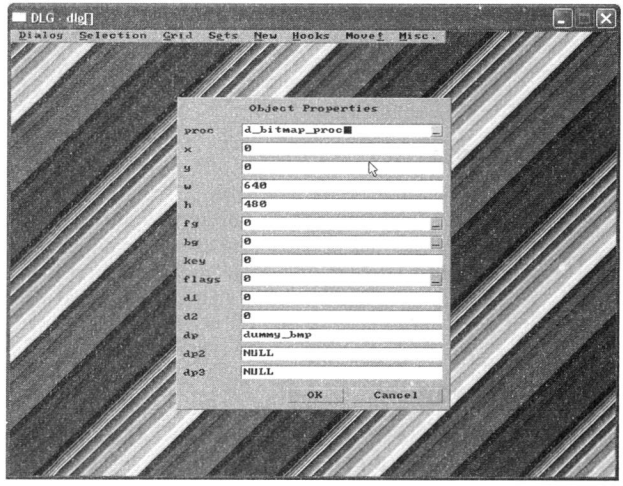


Abbildung 16.2: Das Menühintergrundbild

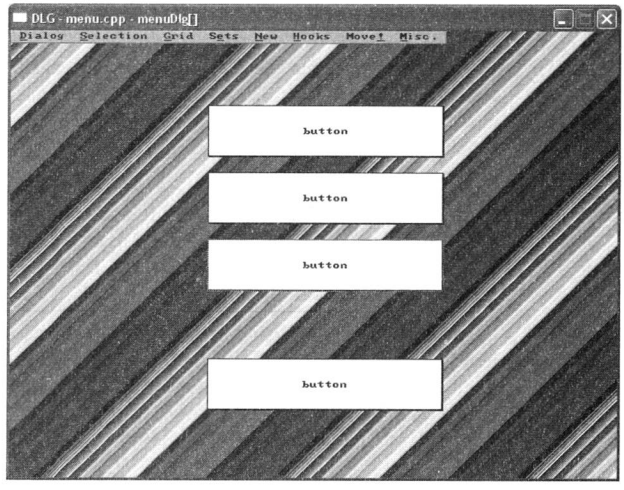


Abbildung 16.3: Der Menü-Dialog in der Vorschau

Die Bestenliste und der Info-Screen

Den Schirm für die Bestenliste und den Info-Screen erstellen Sie ähnlich: Löschen Sie alle Buttons bis auf den untersten. Dann Speichern Sie den Dialog in einer Datei namens »`info.cpp`« und nennen den Dialog »`infoDlg`«.

Und schon sind auch diese zwei Schirme erledigt.

Der Quiz-Screen

Nun geht es ans Eingemachte. Der Quiz-Screen sieht, wenn er fertig ist, in etwas so aus:

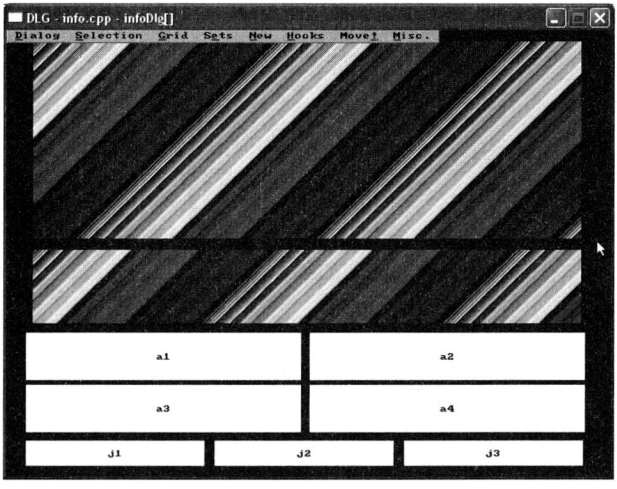


Abbildung 16.4: Der Quiz-Dialog

Aus Übersichtsgründen wurde das Hintergrundbild für den Screenshot entfernt. Wenn Sie es auch vorziehen, die Elemente ohne störendes Hintergrundbild anzuordnen, dann ist das kein Problem. Erstellen Sie erst die Elemente und fügen Sie die Bitmap erst am Ende hinzu. Sie müssen dann allerdings noch die Reihenfolge der einzelnen GUI-Routinen ändern, damit das Hintergrundbild auch zuerst gemalt wird.

Wählen Sie dazu *Selection / List Edit* aus dem Menü, oder drücken Sie einfach [Strg] + [L]. In dem sich nun öffnenden Dialog können Sie mit den Buttons am rechten Rand die Reihenfolge der Elemente beliebig ändern.

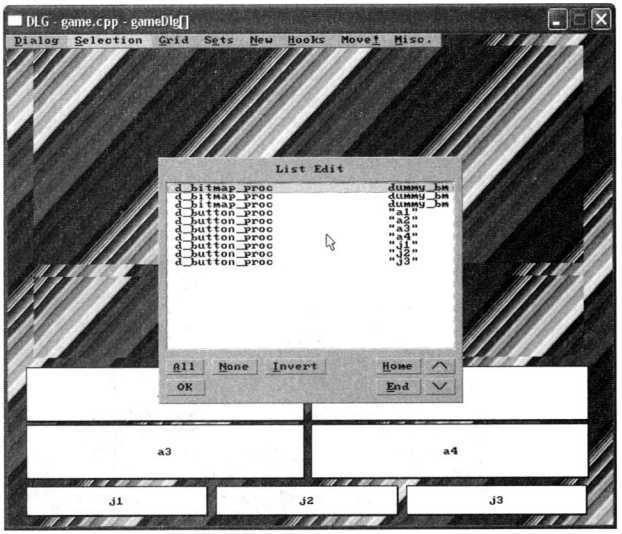


Abbildung 16.5: Reihenfolge der GUI Prozeduren ändern

Wenn Sie alle GUI-Prozeduren in der Reihenfolge anlegen wollen, in der sie nachher auch im Dialog stehen, dann gehen Sie wie folgt vor:

- ✔ Erstellen Sie eine `d_bitmap_proc` mit den Grenzen (0,0, 640, 480) für das Hintergrundbild.
- ✔ Erstellen Sie eine `d_bitmap_proc` mit den Grenzen (28, 12, 580, 212) für eventuell vorhandene Bilder.
- ✔ Erstellen Sie eine `d_bitmap_proc` mit den Grenzen (28, 236, 580, 79) um die Frage darzustellen.
- ✔ Erstellen Sie einen Button mit der Größe (294, 54) und duplizieren Sie ihn 3 mal. Ordnen Sie die Buttons wie in Abbildung 4 gezeigt an.

- ✔ Erstellen Sie einen Button mit der Größe (192, 30) und duplizieren Sie ihn noch zwei mal. Ordnen Sie diese Buttons wie in Abbildung 16.4 gezeigt an.

Wenn Sie möchten, können Sie auch den Text der Buttons leicht ändern, damit Sie sich nachher im erzeugten Code besser zurechtfinden. Bedenken Sie dabei aber, dass DLG von der Englischen Tastaturbelegung ausgehen wird. Ein einfaches Namensschema (wie auf den Screenshots gezeigt) kann Ihnen helfen Verwirrungen zu vermeiden.

Speichern Sie diesen Dialog in einer Datei namens `game.cpp` ab und nennen Sie den Dialog »`gameDlg`«.

Das Hauptprogramm

Nachdem wir bisher nur das Layout entworfen haben, geht es nun daran, das eigentliche Spiel zu erstellen. Das Hauptprogramm ist recht simpel und muss nur dafür sorgen, dass Allegro initialisiert wird. Wir brauchen die Grafik-, Sound-, Timer-, Maus- und Tastaturroutinen.

Der Timer muss von allen anderen Routinen erreichbar sein und wird deshalb in ein eigenes Headerfile ausgelagert.

In dieses Headerfile kommen die Deklarationen aller Variablen und Funktionen, die von mehreren Modulen gebraucht werden. Einige der wichtigsten dieser Funktionen sind unsere Dialogprozedur für den Button und die Funktionen für die einzelnen Programmteile (Menü, Spiel, Info etc.).

Das Hauptprogramm macht eigentlich nicht viel, außer als Bindeglied zwischen den einzelnen Programmpunkten zu fungieren.

Die Button-Prozedur

Der Button wird vollständig als Bitmap gezeichnet. Für die Menüs reichen ein normaler und ein gedrückter Zustand, aber für das eigentliche Spiel brauchen wir noch einige Zustände mehr:

- ✔ Ausgewählt: Hat der Spieler sich für eine Antwort entschieden, dann wird diese in Gelb dargestellt.
- ✔ Korrekt: Ist die Antwort korrekt, dann wird der Button grün.
- ✔ Falsch: Bei einer falschen Antwort färbt sich der Button rot.

Da sowohl der »Falsch«- als auch der »Richtig«-Zustand nur im gedrückten (D_SELECTED) Zustand vorkommen, benutzen wir die d1 Variable, die von der normalen d_button_proc nicht benutzt wird, um diesen Zustand anzuzeigen.

Ist d1 == 0, dann ist der Button nur gedrückt (also gelb). Ist d1 == 1, dann ist der Button gedrückt und die Antwort richtig (also grün). Ist d1 == 2, dann ist der Button gedrückt und die Antwort falsch (also rot).

Wenn die Bilder für die möglichen Zustände in einem Array von BITMAP-Pointern liegen, dann kann die Buttonprozedur in ein paar Zeilen implementiert werden:

```
int button_proc(int msg, DIALOG *d, int c) {
    int w = buttonStates[0]->w;
    int h = buttonStates[0]->h;
    if (msg == MSG_DRAW) {
        if (d->flags & D_SELECTED) {
            if (d->flags & D_EXIT) {
                masked_blit(buttonStates[1],
                        screen, 0, 0, d->x, d->y, w, h);
            } else {
                masked_blit(buttonStates[2+d->d1],
                        screen, 0, 0, d->x, d->y, w, h);
            }
        } else {
            masked_blit(buttonStates[0],
                        screen, 0, 0, d->x, d->y, w, h);
        }
        gui_textout(screen, (const char *)d->dp,
                    d->x+w/2, d->y+h/2, d->fg, TRUE);
        return D_O_K;
    } else {
        return d_button_proc(msg,d, c);
    }
}
```

Die Bilder für die Buttonzustände werden in der init()-Funktion geladen.

```
char* btnNames[] = {"button0.tga", "button1.tga",
                    "button2.tga", "button3.tga",
                    "button4.tga"};
for (int a=0; a < 5; a++) {
    buttonStates[a] = load_bitmap(btnNames[a], NULL);
}
```

Und schon kann der besser aussehende Button benutzt werden.

Das Hauptmenü

Bei der Erstellung des Dialogs in DLG haben Sie bereits die meiste Arbeit erledigt. Nun folgen noch ein paar kleine Anpassungen, wie zum Beispiel das Ändern der verwendeten Dialogprozedur für die Buttons.

Öffnen Sie die Datei `menu.cpp` und stellen Sie sicher, dass die Beschriftung der Buttons (das `dp` Feld) dem festgelegten String entspricht und dass jedem Text ein (`void *`) vorangeht. Das `flags`-Feld der Buttoneinträge muss auf `D_EXIT` stehen, da jeder dieser Buttons das Hauptmenü beendet.

Dann ändern Sie alle `d_button_proc` Einträge in `button_proc` um, damit anstelle des normalen Buttons der Bitmap-Button benutzt wird.

In der `menu()`-Funktion gibt es nur recht wenig zu tun. Das Hintergrundbild muss geladen werden und anstelle der `dummy_bmp` im ersten Eintrag des Dialogs gesetzt werden.

Dann werden die Farben gesetzt, und der Dialog angezeigt.

Der Rückgabewert entspricht dem gedrückten Button +1, da das Hintergrundbild vor allen Schaltflächen im Dialogarray steht.

Wenn wir also den Rückgabewert um eins verringern, dann kommen wir auf den entsprechenden Button. Ein Spezialfall muss noch behandelt werden: Wenn der Spieler auf Esc drückt, dann liefert `do_dialog()` einen negativen Wert. Dies fangen wir ab und liefern stattdessen den Wert zum Beenden des Spieles zurück.

Die ganze Funktion umfasst nur ein paar Zeilen, und doch haben wir ein komplettes Menü.

```
int menu() {
    BITMAP* bg = load_bitmap("title.tga", NULL);

    menuDlg[0].dp = bg;

    show_mouse(screen);
    set_dialog_color(menuDlg,
                makecol(255,255,255), makecol(0,0,0));
    int result = do_dialog(menuDlg, 1);
    if (result <= 0) {
        result = MENU_QUIT;
    } else {
        // Die Position der Buttons entspricht
```

```
            // der Reihenfolge der Menüpunkte
            // allerdings ist vor dem ersten Button
            // noch das Hintergrundbild
            --result;
        }

        destroy_bitmap(bg);
        show_mouse(NULL);

        return result;
    }
```

Den Zustand des Projekts bis zu diesem Zeitpunkt finden Sie auf der CD im Verzeichnis *src/Kapitel16/step01*.

Das Spiel

Jetzt geht es ans Eingemachte. Das eigentliche Spiel besteht auch wieder aus einigen Unterpunkten:

- ✔ Laden der Frage, inklusive Bilder und Sounds
- ✔ Anzeige der Frage
- ✔ Reaktion auf gewählte Antwort/Joker
- ✔ Gegebenenfalls Spielende, ansonsten Anzeige der nächsten Frage

Und in genau dieser Reihenfolge werden wir uns auch an die Implementierung machen.

Laden der Fragen

Die Fragen werden in einer normalen Textdatei gespeichert. Der Aufbau dieser Textdatei muss einem bestimmten Format entsprechen, damit die Fragen korrekt gelesen werden können.

```
Frage: Eine Schwalbe macht noch keinen...
Antwort1: Frühling
Antwort2: Sommer
Antwort3: Herbst
Antwort4: Winter
korrekt: 2
```

```
Frage: Wo befindet sich dieses Bauwerk?
Bild: eiffel.tga
Antwort1: Berlin
Antwort2: Prag
Antwort3: Paris
Antwort4: Helsinki
korrekt: 3

Frage: Was macht dieses Geräusch?
Sound:rasierer.wav
Antwort1: Ein Handy
Antwort2: Eine Türklingel
Antwort3: Ein Auto
Antwort4: Ein Rasierapperat
korrekt: 4

Frage: Wer "A" sagt...
Antwort1: muss auch den B-Test machen
Antwort2: hat Gold im Mund
Antwort3: muss auch B sagen
Antwort4: hat Schmerzen
korrekt: 3

Frage: Welcher Tätigkeit geht "Buffy Summers" nach?
Antwort1: Sie jagt Monster und Dämonen
Antwort2: Sie ist Profi Golferin
Antwort3: Sie ist Detektivin
Antwort4: Sie entschärft Bomben
korrekt: 1
```

Auf das Schlüsselwort `Frage:` folgt der Text, der dann im Spiel innerhalb des Fragekastens steht. Die Antworten kommen dann der Reihe nach in die jeweiligen Boxen. Die richtige Antwort wird durch das Schlüsselwort `korrekt:` identifiziert. Zwischen zwei Fragen steht ein Leerzeichen. Dies sind die Pflichtfelder. Eine Frage kann aber auch noch ein Bild und/oder ein Soundfile enthalten.

Pro Frage kommen wir also auf einen String für die eigentliche Frage, vier Strings für die möglichen Antworten, einen Ganzzahlwert für die Nummer der richtigen Antwort, ein Bild (optional) und ein Soundfile (optional).

```
struct Question {
    char* text;
    char* answer[4];
    int   correct;
```

```
    BITMAP *image;
    SAMPLE* sound;

    Question() {
        text  = NULL;
        for (int a=0; a < 4; a++) {
            answer[a] = NULL;
        }
        image = NULL;
        sound = NULL;
    }
    ~Question() {
        if (image) {
            destroy_bitmap(image);
        }
        if (sound) {
            destroy_sample(sound);
        }
        free(text);
        for (int a=0; a < 4; a++) {
            free(answer[a]);
        }
    }
};

typedef std::list<Question*> QuestionList;
typedef QuestionList::iterator QuestionIterator;
```

Nachdem wir nun eine Struktur für die Fragen haben, können wir die nötigen Routinen schreiben, um sie auch mit Daten zu füllen.

Struktur oder Klasse? Da eine Frage nur der Datenhaltung dient und keine innere Logik benötigt, kann man sie durchaus als Struktur definieren. Aus OOP-Sicht wäre es natürlich besser, so gut wie alles in Objekte zu kapseln, und nur über Methoden auf die Datenelemente zuzugreifen. Und im Falle eines Quizspiels kann man sich nicht einmal mit sonst gerne zur Verteidigung herangezogenen »Performanzgründen« aus der Affäre ziehen. Allerdings hat man in diesem falle außer der Tatsache, sich an die OOP Spielregeln gehalten zu haben keinerlei Vorzüge von der Nutzung einer Klasse. In einem solchen Falle liegt es am Entwickler. Sollten Sie ein strikter Verfechter der Objektorientierung sein, zögern Sie nicht, die Struktur durch eine Klasse zu ersetzen und jeden direkten Datenzugriff auszumerzen.

Die Textdatei mit den Fragen ist zeilenweise aufgebaut. Jede Information steht immer in genau einer Zeile, es gibt weder eine Zeile, in der mehr als eine Information steht, noch eine Information, die sich über mehrere Zeilen ausdehnt.

Diese Eigenschaft können wir uns zu Nutze machen. Um eine komplette Zeile in einen String zu lesen gibt es in der Standard-C-Bibliothek die Routine `fgets()`. Die hat nur einen kleinen Nachteil: Sie speichert das Zeilenendezeichen (`'\n'`) zusammen mit der Zeile ab. Erschwerend kommt hinzu, dass das Zeilenende auf verschiedenen Systemen nicht nur aus verschiedenen Steuerungszeichen besteht, sondern auch die Anzahl der Zeichen, die ein Zeilenende kennzeichnen von Plattform zu Plattform verschieden ist.

Um also auf den »reinen« String zu kommen, ist es am einfachsten, alle nicht druckbaren Zeichen (das sind alle Zeichen, die vor dem Leerzeichen in der ASCII-Tabelle stehen) durch den Stringterminator (`'\0'`) zu ersetzen.

Die Allegro Packfile-Funktionen verhalten sich ebenso wie die Standard-C-Version, können aber mit Zeichenketten in verschiedenen Formaten umgehen, und dabei auch Unterschiede in der Byte Abfolge auf den verschiedenen Systemen ausgleichen.

Es gibt in der Regel für jede Dateizugriffsfunktion der Standard-Bibliothek eine Entsprechung für Allegro Packfiles. Die Allegro Variante ist portabel, das heißt eine Datei, die unter Linux auf einem PowerPC geschrieben wurde, kann mit dem gleichen Programm ohne Probleme auf einem Windows-PC gelesen werden. Zusätzlich können die Packfiles verschlüsselt und komprimiert werden.

Der Code, um eine Zeile einzulesen und die Zeilenendezeichen zu entfernen sieht wie folgt aus:

```
char *readline(char *s, int n, PACKFILE *f) {
    if (pack_fgets(s, n, f) != NULL) {
        char *p = s+ strlen(s);
        while (*p < ' ' && p != s) {
            *p='\0';
            --p;
        }
        return s;
    }
    return NULL;
}
```

Nun brauchen wir »nur« noch die Datei Zeile für Zeile einzulesen. Sobald wir auf eine Zeile mit einer Frage treffen, erzeugen wir ein neue `Question` Struktur und nutzen diese dann zum Speichern aller Folgeinformationen.

Da alle Schlüsselwörter mit verschiedenen Buchstaben beginnen, reicht es in diesem Fall sogar, immer nur das erste Zeichen einer Zeile zu überprüfen, um erkennen zu können, welche Art von Information enthalten ist.

```
Question *loadQuestion(PACKFILE *f) {
    char line[255];
    Question *q = new Question();
    int index = 0;
    int answer = 0;

    char *result = readline(line, 255, f);
    while (result && strlen(line) > 1) {
        switch (toupper(line[0])) {
            case 'F':  // Frage
                // "Frage:" hat 6 Buchstaben
                index = 6;
                while (isspace(line[index])) {
                    ++index;
                }
                q->text = strdup(&line[index]);
                break;

            case 'A':  // Antwort
                answer = line[7] - '1';
                index = 9;
                while (isspace(line[index])) {
                    ++index;
                }
                q->answer[answer] =
                                strdup(&line[index]);
                break;

            case 'K':  // Korrekt
                index = 8;
                while (isspace(line[index])) {
                    ++index;
                }
                // Der index geht von 1-4
                // also müssen wir den Wert um eins
                // verringern
```

```
                q->correct = atoi(&line[index]) -1;
                break;

            case 'B':  // Bild
                index = 5;
                while (isspace(line[index])) {
                    ++index;
                }
                q->image = load_bitmap(&line[index], NULL);
                break;

            case 'S':  // Sound
                index = 6;
                while (isspace(line[index])) {
                    ++index;
                }
                q->sound = load_sample(&line[index]);
                break;
        }
        result = readline(line, 255, f);
    }
    if (q->text) {
        return q;
    }
    delete q;
    return NULL;
}

void freeQuestions() {
    QuestionIterator qi = questions.begin();
    while (qi != questions.end()) {
        delete *qi;
        ++qi;
    }
    questions.clear();
}

void loadQuestions() {
    freeQuestions();
    PACKFILE *f = pack_fopen("fragen.txt", "r");
    Question *q = loadQuestion(f);
    while (q) {
        questions.push_back(q);
        q = loadQuestion(f);
    }
    pack_fclose(f);
}
```

Der Nachteil bei dieser Vorgehensweise ist, dass Fehler im Textfile nicht direkt erkannt werden. Es ist also wichtig, dass der Fragenkatalog korrekt erstellt wird.

Anzeige der Fragen

Den Großteil der Arbeit kann man hier Allegros GUI-Routinen überlassen. Wir müssen nur sicher stellen, dass alle Werte der DIALOG Struktur mit den korrekten Werten gefüllt werden.

Um den folgenden Source Code übersichtlicher zu machen, habe ich ein paar Konstanten für den Zugriff auf das DIALOG-Array definiert.

```
const int BG_IMAGE           = 0;
const int IMAGE_FOR_QUESTION = 1;
const int QUESTION_BOX       = 2;
const int FIRST_ANSWER       = 3;
const int FIRST_JOKER        = 7;
```

Dabei entspricht BG_IMAGE dem Hintergrundbild, IMAGE_FOR_QUESTION ist das zur Frage gehörende Bild und QUESTION_BOX ist die Grafik für den Bereich in dem der Fragetext steht. FIRST_ANSWER ist die Position des ersten Antwort-Buttons im Array, FIRST_JOKER die Position des ersten Joker-Buttons.

Um nun das Dialog-Array in eine gültige Ausgangsposition zu versetzten, müssen alle Zeiger und Flags zurückgesetzt werden.

```
BITMAP *bg         = load_bitmap("bg.tga", NULL);
BITMAP *questionBG = load_bitmap("frage.tga", NULL);
BITMAP *question   = create_bitmap(
                         questionBG->w, questionBG->h);
BITMAP *image      = create_bitmap(
                         gameDlg[1].w, gameDlg[1].h);
BITMAP *joker[4];

joker[0] = load_bitmap("j1.tga", NULL);
joker[1] = load_bitmap("j2.tga", NULL);
joker[2] = load_bitmap("j3.tga", NULL);
joker[3] = load_bitmap("jused.tga", NULL);

set_dialog_color(gameDlg,
                 makecol(255,255,255), makecol(0,0,0));
gameDlg[BG_IMAGE          ].dp = bg;
gameDlg[IMAGE_FOR_QUESTION].dp = image;
```

```
gameDlg[QUESTION_BOX      ].dp = question;

for (int a=0; a < 3; a++) {
    gameDlg[FIRST_JOKER + a].dp    = joker[a];
    gameDlg[FIRST_JOKER + a].flags = D_EXIT;
}
```

Sobald nun die Fragen geladen wurden, kann man auch die Texte der Antwort-Buttons setzen und gegebenenfalls das zur Frage gehörige Bild anzeigen.

```
loadQuestions();
QuestionIterator itor = questions.begin();
Question *q = *itor;
for (int a=0; a < 4; a++) {
    gameDlg[FIRST_ANSWER+a].dp    = q->answer[a];
    gameDlg[FIRST_ANSWER+a].d1    = 0;
    gameDlg[FIRST_ANSWER+a].flags = D_EXIT;
}

blit(bg, image,
         gameDlg[IMAGE_FOR_QUESTION].x,
         gameDlg[IMAGE_FOR_QUESTION].y,
         0, 0,
         gameDlg[IMAGE_FOR_QUESTION].w,
         gameDlg[IMAGE_FOR_QUESTION].h);

if (q->image) {
    blit(q->image, image, 0, 0,
         (image->w - q->image->w)/2,
         (image->h - q->image->h)/2,
         q->image->w,
         q->image->h);
}
if (q->sound) {
    play_sample(q->sound, 255, 128, 1000, 0);
}
```

Zuerst werden die Texte, der Status und die Flags der Antwort-Buttons gesetzt. Dann wird der entsprechende Teil des Hintergrundbildes auf das Fragebild kopiert (damit im Falle eines zu kleinen Fragebildes kein unschöner Rand zu sehen ist).

Hat die Frage ein Bild gesetzt, so wird es zentriert in seinem Bereich angezeigt, ist ein Sound gesetzt, so wird er abgespielt.

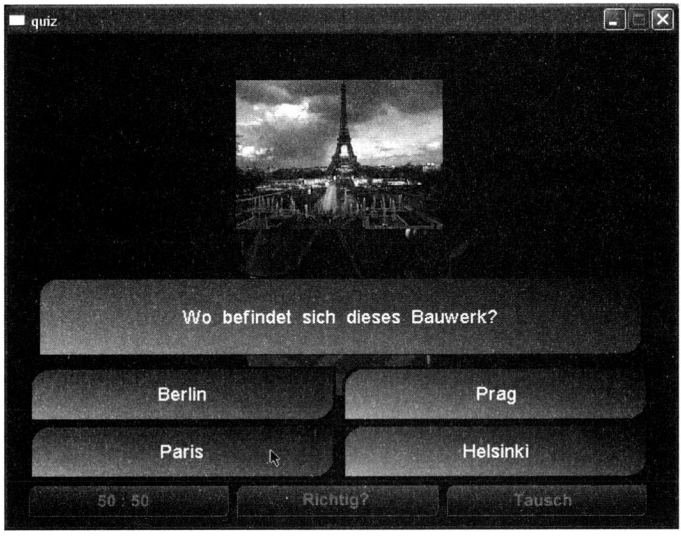


Abbildung 16.6: Quizfrage mit zugehörigem Bild

Spielablauf

Die Frage wird durch einen Aufruf von `do_dialog()` angezeigt. Je nachdem, ob der Nutzer nun versucht die Frage zu lösen oder einen Joker benutzt, verzweigt der Programmablauf.

Hat der Spieler den »50:50-Joker« benutzt, werden nach der obligatorischen Sicherheitsabfrage zwei falsche Antworten rot markiert und können ab da auch nicht mehr als Antwort angewählt werden.

Beim »Richtig?-Joker« wird ein internes Flag gesetzt, das dann beim Klick auf eine Antwort überprüft wird.

Beim »Tausch-Joker« wird zur nächsten Frage weitergesprungen.

Klickt der Spieler auf eine Antwort, so wird diese als gedrückt markiert und dann wird `do_dialog()` ein weiteres Mal aufgerufen. Dieser Aufruf ist notwendig, damit sich die Elemente auch neu zeichnen können. Um sicher zu stellen, dass dieser Zustand nur für kurze Zeit angezeigt wird,

wird ein Time-Out definiert. Eine spezielle Dialogprozedur, die den Dialog beendet, sobald ein Time-Out festgestellt wird, muss aus diesem Grund am Ende des Dialogs eingefügt werden.

Schließlich wird die Antwort als korrekt oder falsch markiert. Ist sie korrekt, wird die nächste Frage angezeigt. Ist sie falsch, dann wird erst überprüft, ob der »Richtig?-Joker« derzeit verwendet wird. Ist dies der Fall, kann der Nutzer sich noch zwischen den anderen drei Antworten entscheiden. Wenn nicht, so wird das Spiel beendet und die erreichte Punktzahl zurückgegeben.

```
while (!quit && ! nextQuestion) {
    int index = do_dialog(gameDlg, -1);
    if (index < 0) {
        goto done;
    }
    if (index >= FIRST_JOKER) {
        gameDlg[index].flags = D_SELECTED | D_DISABLED;
        gameDlg[index].dp    = (void*) joker[3];
        int a1, a2, ok;
        switch (index - FIRST_JOKER) {
            case 0: // 50:50
                ok = message("Sind Sie sicher, dass Sie den\n50:50
               Joker wollen?");
                if (ok) {
                    a1 = rand() % 4;
                    while (a1 == q->correct || (gameDlg[FIRST_ANSWER
                   + a1].flags  & D_DISABLED)) {
                        a1 = rand() % 4;
                    }
                    a2 = rand() % 4;
                    while (a2 == q->correct || a2 == a1  ||
                    (gameDlg[FIRST_ANSWER + a2].flags  &
                    D_DISABLED)) {
                        a2 = rand() % 4;
                    }
                    gameDlg[FIRST_ANSWER + a1].flags = D_DISABLED +
                    D_SELECTED;
                    gameDlg[FIRST_ANSWER + a1].d1    = 2;
                    gameDlg[FIRST_ANSWER + a2].flags = D_DISABLED +
                  D_SELECTED;
                    gameDlg[FIRST_ANSWER + a2].d1    = 2;
                }
                break;
            case 1: // richtig?
                ok = message("Sind Sie sicher, dass Sie den\nRichtig?
               Joker wollen?");
```

```
                if (ok) {
                    check = TRUE;
                }
                break;
            case 2: // tauschen
                nextQuestion = TRUE;
                break;
        }
    } else {

        answered     = TRUE;
        timeOut = timerCounter + 60;
        gameDlg[index].flags = D_SELECTED;
        do_dialog(gameDlg, -1);

        if (q->correct+FIRST_ANSWER == index) {
            gameDlg[index].dl = 1;
            timeOut = timerCounter + 60;
        } else {
            gameDlg[index].dl = 2;
            if (check) {
                timeOut = timerCounter + 1;
            } else {
                timeOut = timerCounter + 60;
            }

        }

        do_dialog(gameDlg, -1);

        if (q->correct+ FIRST_ANSWER == index) {
            nextQuestion = TRUE;
        } else if (check) {
            check = FALSE;
            gameDlg[index].flags =
                   D_SELECTED | D_DISABLED;
            gameDlg[index].dl    = 2;

        } else {
            quit = TRUE;
        }

    }
    answered = FALSE;
}
++itor;
```

Die Bestenliste

Um die Rangliste verwalten zu können, brauchen wir zuerst eine Struktur (oder Klasse, je nach Vorliebe), um die Ergebnisdaten zu speichern. Jeder Spieler, der sich in der Bestenliste verewigen darf, kann seinen Namen eingeben (acht Zeichen maximal). Dieser wird dann zusammen mit der erreichten Punktzahl gespeichert.

```
const int HISCORE_LIST_SIZE = 10;
struct HiScoreEntry {
    int score;
    char name[9];
};

extern HiScoreEntry hiScoreList[HISCORE_LIST_SIZE+1];
```

Wir speichern zehn Einträge in der Liste. Allerdings enthält die Liste einen Eintrag mehr als notwendig. Der Grund hierfür ist simpel: Dieser extra Eintrag macht die Implementierung der Liste deutlich einfacher. Denn um nun einen neuen Eintrag zur Liste hinzuzufügen, können wir den Namen des Spielers und seine Punktzahl im 11ten Eintrag speichern, dann die Liste mit der Standard-C-Funktion `qsort()` sortieren und haben die besten zehn Ergebnisse in den ersten `HISCORE_LIST_SIZE` Stellen.

Die Funktion `qsort()` bekommt als Parameter das zu sortierende Array, die Anzahl der Einträge, die Größe eines Eintrags und einen Zeiger auf eine Funktion, welche zwei Einträge vergleichen kann.

```
// Vergleicht die 2 übergebenen Argumente
int cmp(const void* lhs, const void* rhs) {
    return (((HiScoreEntry*) lhs)->score -
            ((HiScoreEntry*) rhs)->score) * (-1);
}
    // Und an anderer Stelle im Code
    // Wird dann sortiert
    qsort(hiScoreList,          // Array
          HISCORE_LIST_SIZE+1, // Anzahl Elemente
          sizeof(HiScoreEntry),// Größe eines Elements
          cmp);                // Vergleichsfunktion
```

Bei der `cmp()`-Funktion ist eigentlich nur der Rückgabewert entscheidend. Ein negativer Wert bedeutet, dass der erste Eintrag kleiner ist als der zweite, ein positiver Wert bedeutet, dass der erste Eintrag größer ist als der zweite. Wird 0 zurückgegeben, sind beide Werte gleich.

Allerdings sortiert qsort() in aufsteigender Reihenfolge (auf dem ersten Platz ist der kleinste Wert, auf dem letzten Platz der höchste), eine High-Score Liste ist allerdings absteigend sortiert (der höchste Wert auf Platz eins, der geringste Wert auf dem letzten Platz). Aus diesem Grund wird das Vorzeichen des Ergebnisses vor der Rückgabe noch negiert (mit –1 multipliziert). Jetzt wird die Liste – wie benötigt – absteigend sortiert.

Laden und Speichern

Beim Laden und Speichern greifen wir wieder auf die Allegro-Packfile-Routinen zurück. Beachten Sie, dass die Punktzahl mit der Funktion pack_iputl() auf die Platte geschrieben bzw. mit pack_igetl() wieder gelesen wird. Diese Funktionen schreiben/lesen einen Ganzzahlwert mit vier Bytes Länge im Intel-Format und konvertieren diesen Wert in das auf dem aktuellen Rechner gültige Zahlenformat.

```
void loadScores() {
    if (!exists("hiscore.dat")) {
        for (int a=0; a <  HISCORE_LIST_SIZE +1; a++) {
            hiScoreList[a].name[0] = '\0';
            hiScoreList[a].score   = 0;
        }
        return;
    }
    PACKFILE *f = pack_fopen("hiscore.dat", "rb");
    for (int a=0; a <  HISCORE_LIST_SIZE; a++) {
        pack_fread(hiScoreList[a].name , 9, f);
        hiScoreList[a].score =  pack_igetl(f);
    }
    pack_fclose(f);
}
void storeScores() {
    PACKFILE *f = pack_fopen("hiscore.dat", "wb");
    for (int a=0; a <  HISCORE_LIST_SIZE; a++) {
        pack_fwrite(hiScoreList[a].name, 9, f);
        pack_iputl (hiScoreList[a].score, f);
    }
    pack_fclose(f);
}
```

Für den Fall, dass beim Laden noch keine Bestenliste auf Platte vorhanden sein sollte, werden alle Werte mit 0 initialisiert.

Anzeigen der Liste

Die `showScores()`-Funktion hat zwei Möglichkeiten die Liste anzuzeigen:

- ✔ Wenn `waitForInput` gleich 0 ist, wird die Liste direkt auf den `screen` geschrieben, dann kehrt die Funktion sofort zurück.
- ✔ Wenn `waitForInput` gleich 1 ist (der Defaultwert), dann wird die Liste im Rahmen eines Dialoges angezeigt. Um den Dialog zu beenden, muss der OK-Button gedrückt werden.

```
// In quiz.h
void showScores(int waitForInput=1);

// In hiscore.cpp
void showScores(int waitForInput) {
    BITMAP *bg  = load_bitmap("plainbg.tga", NULL);
    BITMAP *btn = load_bitmap("ok.tga"     , NULL);

    loadScores();

    int y = 130;
    for (int a=0; a < 10; a++) {
        textprintf_right (bg, font, 200, y+20*a,
                  makecol(255, 255, 255), "%i", (a+1));
        textprintf_centre(bg, font, 320, y+20*a,
                   makecol(255, 255, 255), "%s",
                   hiScoreList[a].name);
        textprintf_right (bg, font, 440, y+20*a,
                   makecol(255, 255, 255), "%i",
                   hiScoreList[a].score);
    }

    infoDlg[0].dp = (void*) bg;
    infoDlg[1].dp = (void*) btn;

    if (waitForInput) {
        do_dialog(infoDlg, -1);
    } else {
        scare_mouse();
        blit(bg, screen, 0, 0, 0, 0, bg->w, bg->h);
        unscare_mouse();
    }

    destroy_bitmap(bg);
    destroy_bitmap(btn);
}
```

Neuen Eintrag hinzufügen

Mit all der Vorarbeit ist das Hinzufügen eines neues High Scores kein Problem mehr. Wir prüfen, ob die Punktzahl hoch genug ist, um in die Liste aufgenommen zu werden. Wenn dem so ist, kann der Spieler seinen Namen eingeben, der Eintrag wird hinzugefügt und die Liste auf Platte gespeichert.

Von diesen Schritten ist das Eingeben des Namens mit dem meisten Programmieraufwand verbunden, da wir eine neue Dialogbox mit einem Eingabefeld benötigen.

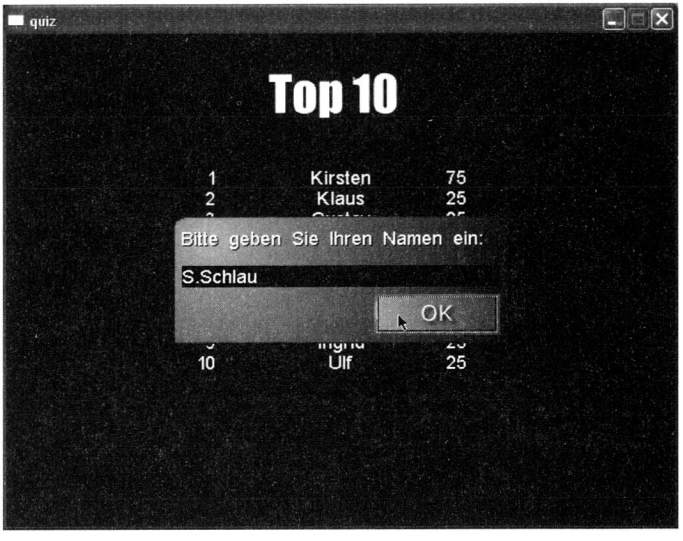


Abbildung 16.7: Der Eingabedialog für den Namen

Erzeugen Sie mit DLG eine `d_bitmap_proc` der Größe 320x124 und platzieren Sie ein Eingabefeld (`d_edit_proc`) an (8,48) mit der Größe 304x8. Zum Abschluss brauchen wir noch einen Bitmap-Button (`d_icon_proc`) an der Position (192, 176) mit der Größe 120x40. Speichern Sie das Ergebnis in `msg.cpp` ab, und nennen Sie den Dialog `inputDlg[]`.

Die d_edit_proc speichert einen Zeiger auf den Ihr zu Verfügung stehenden Speicher in dp. Die maximale Anzahl an Zeichen, die eingegeben werden dürfen, steht in d1, die aktuelle Cursorposition in d2.

Die input()-Funktion kopiert dann die eingegebenen Zeichen vom Speicher der Dialogprozedur in den übergebenen Speicherbereich (in unserem Fall: Die High Score Liste).

```
void input(char *msg, char *buf, int max) {
    BITMAP *bg  = load_bitmap("input.tga", NULL);
    BITMAP *btn = load_bitmap("ok.tga"   , NULL);

    char buffer[12];
    buffer[0] = 0;

    inputDlg[0].dp = (void*) bg;
    inputDlg[1].dp = (void*) buffer;
    inputDlg[2].dp = (void*) btn;

    inputDlg[1].d1 = MIN(11, max);
    inputDlg[1].d2 = 0;

    set_dialog_color(inputDlg,
                     makecol(255,255,255),
                     makecol(0,0,0));
    centre_dialog(inputDlg);

    textout(bg, font, msg, 8, 12, makecol(0,0,0));
    textout(bg, font, msg, 7, 11, makecol(255, 255, 255));

    int result = do_dialog(inputDlg, 1);

    if (max < 12) {
        buffer[max] = '\0';
    }
    strcpy(buf, buffer);


    destroy_bitmap(bg);
    destroy_bitmap(btn);
}
```

Im letzten Schritt müssen diese Funktionen auch vom Hauptprogramm aufgerufen werden:

```
// in quiz.cpp
    while (!gameOver) {
        switch (menu()) {
            case MENU_START_GAME:
                score = game();
                appendScore(score);
                break;
            case MENU_HISCORE:
                showScores();
                break;
            case MENU_QUIT:
                gameOver = TRUE;
                break;
        }
    }
```

Die Programm-Informationen

Zum Abschluss die leichteste Übung: Das Anzeigen der Bitmap mit den Hinweisen auf Programmierer, Urheberrecht und ähnlichen Informationen. Wir benutzen zu diesem Zweck den gleichen Dialog wie zum Anzeigen der Bestenliste:

```
void showInfo() {
    BITMAP *bg  = load_bitmap("info.tga", NULL);
    BITMAP *btn = load_bitmap("ok.tga"     , NULL);

    infoDlg[0].dp = (void*) bg;
    infoDlg[1].dp = (void*) btn;

    do_dialog(infoDlg, -1);

    destroy_bitmap(bg);
    destroy_bitmap(btn);
}
```

So können Sie die Informationen nach Belieben präsentieren.

Den kompletten Quellcode finden Sie wie immer auf der beliegenden CD.

Übungen

- ✔ Verändern Sie den Quellcode so, dass die Fragen in einer zufälligen Reihenfolge gestellt werden.
- ✔ Benutzen Sie die Packfile-Kompression für Ihre Fragen und Bestenliste.
- ✔ Fügen Sie ein Kommando hinzu, um auch MIDI-Dateien zu einer Frage abspielen zu können.
- ✔ Stellen Sie sicher, dass Musik- und Sound-Dateien aufhören zu spielen, wenn der Nutzer die Frage beantwortet hat.

17 Sprites

In diesem Kapitel geht es um Sprites. Sprites in allen Variationen. Wir werden sie drehen, spiegeln, strecken, stauchen und noch andere interessante Dinge mit ihnen machen.

Der Name *Sprite* kommt aus der Heimcomputer- und Spielkonsolenzeit. Sie bezeichnen kleine Grafiken, in denen Pixel mit einem bestimmten Farbwert nicht angezeigt werden. Diese wurden damals direkt von der Hardware auf den Bildschirm gebracht, und niemand musste sich Gedanken darüber machen, was sich unter den Sprites befand. Ein gutes Beispiel für ein Sprite im eigentlichen Sinne ist zum Beispiel der Mauszeiger. Obwohl er eigentlich rechteckig ist, wird durch die transparenten Pixel der Umriss eines Zeigers erzeugt. Auch hier muss sich der Programmierer nicht darum kümmern, den Hintergrund unter der Maus wiederherzustellen.

Auf dem Amiga, einem populären Heimcomputer, wurden Sprites als *Bob* bezeichnet (für ***Blitter Object***). Wundern Sie sich also nicht, wenn Sie irgendwo auf diesem Begriff stoßen. Ein Bob ist ein Sprite.

Auf dem PC gab es diese Hardwareunterstützung nicht. Also fingen die Entwickler an, die entsprechende Funktionalität selbst zu entwickeln. Sie legten eine bestimmte Farbe (oder im 8-Bit-Modus einen bestimmten Index der Farbpalette) fest, die von den Anzeigeroutinen einfach nicht beachtet werden sollte. Auf diese Weise konnten Sie einen ähnlichen Effekt erzielen, mussten aber den Hintergrund selbst wiederherstellen.

Arten von Sprites

Allegro unterstützt verschiedene Spritevarianten. Sie alle haben ihre Vor- und Nachteile und deswegen betrachten wir sie einmal ganz genau.

Normale Bitmaps

Jede Bitmap kann als Sprite verwendet werden. Wird die Grafik mit `masked_blit()` anstatt von `blit()` angezeigt, dann werden Bereiche die mit der Farbe Pink (RGB: 255,0,255) gefüllt sind nicht beachtet. Im 8-

Bit-Modus wird der Index 0 von der Routine übersprungen und erfüllt den gleichen Effekt. Neben `masked_blit()` können Sie auch die `draw_sprite()`-Methoden benutzen, die Ihnen in einigen Fällen etwas Schreibarbeit ersparen können, da sie weniger Parameter benutzen. Diese Art von Sprites ist am flexibelsten. Sie können sie drehen, spiegeln und auf sie malen.

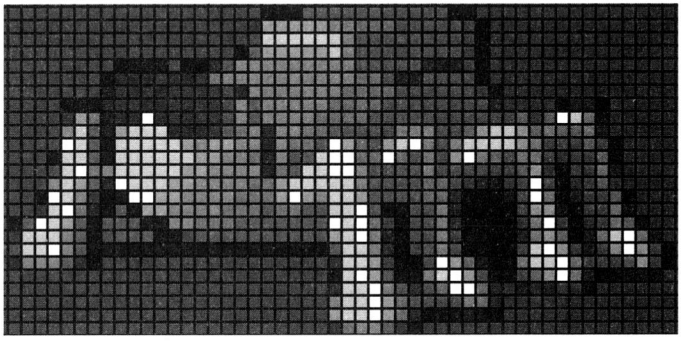


Abbildung 17.1: Ein simples Sprite

Sie können sich vorstellen, dass die Methode zum Darstellen dieser Sprites in etwa so aussieht:

```
Für alle Reihen
    Für alle Spalten
        Ist der Pixel transparent?
            Ja: tue nichts
            Nein: Pixel anzeigen
```

Die tatsächliche Routine ist deutlich komplexer, nimmt Rücksicht auf Spezialfälle und hat auch einiges an Optimierungen eingebaut.

Memory-Bitmaps

Normalerweise werden alle Bitmaps (bis auf `screen`) im Hauptspeicher angelegt. Dies ist in etwa so, als wenn Sie einen entsprechend großen Speicherblock mit `malloc()`, `calloc()` oder `new` anlegen würden.

Sie können diese Art von Bitmaps entweder mit Hilfe der bereits besprochenen Funktionen laden oder mit einer dieser Funktionen erzeugen:

Funktionsname	Beschreibung
`BITMAP *create_bitmap (int w, int h)`	Erzeugt eine Bitmap mit der angegebenen Breite (`w`) und Höhe (`h`) in der derzeit gesetzten Farbtiefe.
`BITMAP *create_bitmap_ex(int depth, int w, in h)`	Wie `create_bitmap()`, aber diesmal wird die Farbtiefe der neuen Bitmap auf den Wert `depth` gesetzt.

Tabelle 17.1: Funktionen zum Erzeugen einer Memory-Bitmap

Sub-Bitmaps

Sub-Bitmaps sind eigentlich keine »richtigen« Bitmaps. Sie definieren einen Bereich in einer anderen Bitmap. Stellen Sie sich vor, Sie laden eine Bitmap, in der alle Animationsphasen Ihrer Spielfigur enthalten sind. Nun könnten Sie für jede einzelne Animationsphase eine Sub-Bitmap anlegen, und dann diese so behandeln, als ob Sie jeden Frame in einer eigenen Bitmap haben.

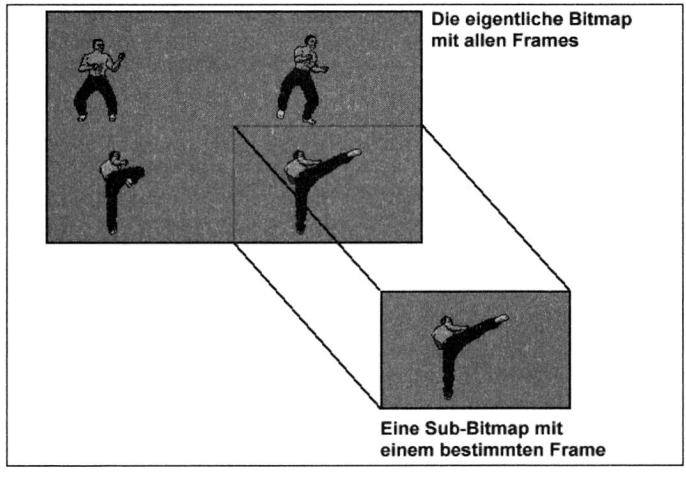


Abbildung 17.2: Eine Bitmap mit einer Sub-Bitmap

In dem in Abbildung 17.2 dargestellten Fall könnten Sie auch immer einen bestimmten Teil der Bitmap mittels `masked_blit()` ausgeben. Allerdings macht eine Sub-Bitmap die Berechnung überflüssig, und sobald Sie Sprites rotieren oder spiegeln wollen, führt kein Weg mehr daran vorbei, dass Sie für einen einzelnen Animationsframe auch eine eigene Bitmap brauchen.

Funktionsname	Beschreibung
`BITMAP *create_sub_bitmap(BITMAP *parent, int x, int y, int width, int height);`	Erzeugt eine Sub-Bitmap, die den angegebenen Teil der übergebenen Bitmap darstellt. Die angegebene Position muss zwingend innerhalb der Ursprungsbitmap liegen, falls ein Teil des durch `width` und `height` angegebenen Bereichs außerhalb des Ursprungsbilds liegt, dann wird die Größe automatisch angepasst.

Tabelle 17.2: Erstellen einer Sub-Bitmap

Video-Bitmaps

Video-Bitmaps werden direkt im Speicher der Grafikkarte angelegt. Dies hat einige Vorzüge, zum Beispiel den, dass das Blitten von einer Video-Bitmap auf eine andere Video-Bitmap sehr schnell geht, da diese Operation meistens von der Grafikkarte beschleunigt wird.

Allerdings werden in der Regel nur direkte Blits und transparente (also Sprite-Blits) von der Hardware beschleunigt. Rotation und Spiegelung wird immer noch in Software durchgeführt.

Behalten Sie immer im Kopf, dass Lesezugriffe auf Video-Bitmaps langsam sind. Alle Aktionen, die es nötig machen, die ursprünglichen Pixel erst zu lesen, werden dadurch verlangsamt.

Sollten Sie also manche Sprites häufig rotieren, halb-transparent darstellen oder spiegeln wollen, dann sollten Sie diese wie bisher auch als normale Memory-Bitmap handhaben.

Funktionsname	Beschreibung
`BITMAP *create_video_bitmap(BITMAP int width, int height);`	Erzeugt eine Video-Bitmap der angegeben Größe. Diese Bitmap kann den gleichen Bereich wie die globale `screen` Variable belegen. Aus diesem Grund sollten Sie zuerst immer einen Bitmap mit der Größe der aktuellen Auflösung anfordern.

Tabelle 17.3: Erstellen einer Video-Bitmap

System-Bitmaps

System-Bitmaps sind so eine Art Mischform zwischen Memory-Bitmaps und Video-Bitmaps. Sie liegen in einem Speicherbereich, auf den sowohl die Grafikkarte als auch der Hauptspeicher Zugriff haben. Wenn Sie also lesend auf den Speicher zugreifen müssen, aber dennoch Beschleunigung haben wollen, dann ist der System Speicher die ideale Position für Ihre Bitmap. Zwar ist der Zugriff seitens der Grafikkarte nicht so schnell wie beim Videospeicher, aber der deutlich schnellere Lesezugriff macht dies wieder wett.

Funktionsname	Beschreibung
`BITMAP *create_system_bitmap(BITMAP int width, int height);`	Erzeugt eine System-Bitmap der angegeben Größe. Kann keine System-Bitmap erzeugt werden, dann wird eine Memory-Bitmap zurückgegeben.

Tabelle 17.4: Erstellen einer System Bitmap

Die Hardware fragen

Es ist auch wichtig, dass Sie überprüfen, ob eine bestimmte Variante von Bitmap Blits durch die Hardware beschleunigt wird. Wenn Sie alle Ihre Grafiken im Videospeicher hinterlegen, aber die Grafikkarte keine Unterstützung für das Anzeigen dieser hat, dann ginge der Schuss nach hinten los.

Wir beschäftigen uns in diesem Kapitel noch ausführlich mit den Möglichkeiten ein Spiel so schnell wie möglich zu machen. Wenn Sie es schon jetzt vor Neugierde nicht mehr aushalten, dann starten Sie das Programm »`test.exe`« (oder »`test`« auf Linux-Systemen) im Allegro »tests« Ordner (z.B.: `/usr/home/coder/allegro/tests`, oder `c:\allegr\tests`).

Suchen Sie sich einen Grafikmodus aus und bestätigen Sie. Wählen Sie dann den Menüpunkt `Misc / Accelerated features`. Sie sollten eine Auflistung erhalten, die in etwa so aussieht:

```
                    Hardware accelerated features

                          solid hline:  yes
                            xor hline:  no
        solid/masked pattern hline:  no
                 copy pattern hline:  no
                           solid fill:  yes
                             xor fill:  no
         solid/masked pattern fill:  no
                 copy pattern fill:  no
                           solid line:  no
                             xor line:  no
                      solid triangle:  no
                        xor triangle:  no
                            mono text:  no
                     vram->vram blit:  yes
           masked vram->vram blit:  yes
                    mem->screen blit:  no
          masked mem->screen blit:  no
                 system->screen blit:  yes
        masked system->screen blit:  yes

                    Press a key or mouse button
```

Abbildung 17.3: Beschleunigte Funktionen

Wie Sie sehen kann meine aktuelle Grafikkarte das Füllen von Flächen und Linien beschleunigen (wenn diese Operationen innerhalb des Grafikspeichers ablaufen), und alle Arten von Blits, außer denen von Hauptspeicher zu Video Speicher. Bei der Geschwindigkeit, mit der sich die Beschleunigerkarten entwickeln, können Sie davon ausgehen, dass die meisten zumindest die Video RAM/Video RAM Blits beschleunigen werden.

RLE-Sprites

Wenn Sie auf Drehungen und Spiegelungen verzichten können oder es für Sie kein Problem ist, für alle Varianten (also gedreht und gespiegelt) eigene Sprites anzulegen, dann können *RLE-Sprites* einen gewaltigen Geschwindigkeitsvorteil bringen. RLE steht für ***Run Length Encoding***. Bei dieser Methode wird gezählt wie viele transparente Pixel aufeinander folgen. Diese Menge von Pixels wird als Run bezeichnet.

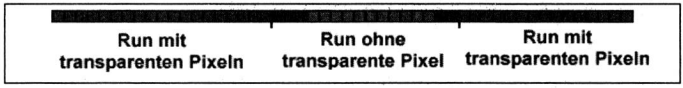


Abbildung 17.4: Eine Zeile in einem RLE-Sprite

Wenn nun dieses Sprite abgebildet werden soll, dann ist es nicht mehr notwendig jeden einzelnen Pixel zu überprüfen. Man kennt die Anzahl der transparenten Pixel die aufeinander folgen, und kann alle auf einmal überspringen. Und die sichtbaren Bereiche können ebenfalls mit einer einzigen Kopieraktion an ihr Ziel gebracht werden.

Da die Daten nun in einem speziellen Format gespeichert werden, kann man nicht mehr mit den normalen Sprite-Funktionen auf sie zugreifen, sondern man benötigt einen neuen Satz von Funktionen:

Funktionsname	Beschreibung
`RLE_SPRITE* get_rle_sprite(BITMAP* bmp)`	Erzeugt ein RLE-Sprite mit den Daten der übergebenen Bitmap. Diese Bitmap muss eine Memory-Bitmap sein.
`destroy_rle_sprite(RLE_SPRITE* sprite)`	Gibt den vom Sprite belegten Speicher wieder frei.
`draw_rle_sprite (BITMAP* dest, RLE_SPRITE* sprite, int x, int y)`	Zeigt das Sprite an der angegebenen Position auf der Ziel Bitmap an.

Funktionsname	Beschreibung
`draw_trans_rle_sprite (BITMAP* dest, RLE_SPRITE* sprite, int x, int y)`	Zeigt eine durchscheinende Version des Sprites an.
`draw_lit_rle_sprite (BITMAP* dest, RLE_SPRITE* sprite, int x, int y)`	Zeigt eine eingefärbte Version des Sprites an.

Tabelle 17.5: RLE Sprite-Funktionen

Compiled Sprites

Kompilierte Sprites waren auf älteren Rechnern das so ziemlich schnellste, das möglich war. Anstatt die Bilddaten zu speichern, wurde ein spezieller Code erzeugt, der das Bild Pixel für Pixel gemalt hat.

Auf heutigen Rechnern ist der Unterschied jedoch kaum noch bemerkbar. Und da diese Art von Sprites noch weniger Flexibilität bietet als RLE-Sprites (Sie können diese Art von Sprites wirklich nur anzeigen) wird sie heute kaum noch benutzt. Wenn Sie jedoch einmal ein Spiel für 486er Rechner optimieren wollen, dann könnten diese Sprites ein paar Frames mehr Geschwindigkeit bringen. Da 486er heute jedoch kaum noch im Einsatz sind, werde ich hier nicht weiter auf kompilierte Sprites eingehen.

Sprites erzeugen

Nachdem nun die Grundlagen geklärt sind, können wir anfangen Sprites zu benutzen. Im einfachsten Fall ist ein Sprite einfach eine Bitmap, in der die transparenten Bereiche mit magischen Pink (RGB: 255,0,255) gefüllt sind.

Als Beispiel-Sprite nutzen wir einen netten Pinguin, der aus einigen wenigen geometrischen Grundkörpern in einem 3D-Programm erstellt wurde. In einem ersten Versuch lassen wir dieses Sprite von links nach rechts über den Bildschirm schweben. Wenn Sie der Meinung sind, dass dies ja

wohl eines der leichtesten Beispiele ist, haben Sie recht. Wir werden aber ausgehend von diesem Programm noch einige andere kleine Änderungen einfügen und das Beispiel dadurch interessanter machen.

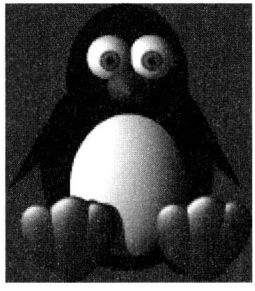


Abbildung 17.5: Ein Beispiel-Sprite

Das Sprite wird auf bekannte Weise geladen:

```
BITMAP *sprite = load_bitmap("pingu.tga", NULL);
```

Das Sprite soll sich in vier Sekunden einmal von links nach rechts bewegt haben. Aus diesem Grund berechnen wir die Geschwindigkeit wie folgt:

```
    const float ticks = 4.0 * 60.0;

    float  x = -sprite->w;
    float  y = (SCREEN_H - sprite->h) / 2;

    float dx = SCREEN_W / ticks;
    float dy = SCREEN_H / 4.0 / ticks * -1.0;
```

Wir greifen wieder auf die Hilfsmethoden aus der `util.cpp` zurück, damit das Hauptprogramm übersichtlich bleibt.

```
#include <allegro.h>
#include "util.h"

int main(int argc, char** argv) {
    // Grafik modus. 640x480,
    // Logik Timer auf 60 fps
    init(640, 480, 60);
```

```
    BITMAP *sprite = load_bitmap("pingu.tga", NULL);

    const float ticks = 4.0 * 60.0;
    float x  = -sprite->w;
    float y  = (SCREEN_H - sprite->h) / 2;
    float dx = SCREEN_W / ticks;
    float dy = SCREEN_H / 4.0 / ticks * -1.0;


    int needsRepaint = TRUE;

    syncTimer(&timerCounter);
    while (!key[KEY_ESC]) {
        if (timerCounter > 0) {
            while (timerCounter > 0) {
                x += dx;
                y += dy;
                --timerCounter;

                if (x > SCREEN_W) {
                    x = -sprite->w;
                    y = (SCREEN_H - sprite->h) / 2;
                }
            }
            // Hintergrund löschen
            clear_to_color(doubleBuffer, makecol(0,0, 230));
            // Sprite zeichnen
            draw_sprite(doubleBuffer, sprite, (int) x, (int) y);
            needsRepaint = TRUE;
        }

        if (needsRepaint) {
            needsRepaint = FALSE;
            show();
        }
    }
    destroy_bitmap(sprite);
    done();
    return 0;
} END_OF_MAIN();
```

Nachdem nun das Gerüst steht, werden wir den Grundstein für schnelle Grafiken legen. Wir ändern die `init()`-Methode ab, damit der `doubleBuffer` nach Möglichkeit im Speicher der Grafikkarte erzeugt wird.

```
    /* Den DoubleBuffer erstellen */
    screenAlias  = create_video_bitmap(
                        SCREEN_W, SCREEN_H);
    doubleBuffer = create_video_bitmap(
                        SCREEN_W, SCREEN_H);
    if (!doubleBuffer) {
        if (screenAlias != NULL) {
            destroy_bitmap(screenAlias);
        }
        doubleBuffer = create_system_bitmap(
                        SCREEN_W, SCREEN_H);
        if (!doubleBuffer) {
            fatalError("Unable to create double buffer");
        }
    }
```

Da sich die screen-Variable den Speicher mit den angeforderten Bitmaps teilt, ist es wichtig, erst einen Bereich von der Größe des Schirms anzufordern. Diese Bitmap, die sich den Speicher mit dem screen teilt, hat den Namen screenAlias. Sie wird fürs erste stillschweigend mit geschleppt und erst in der done()-Methode wieder freigegeben.

Wenn wir nicht in der Lage waren einen doubleBuffer zu erzeugen, dann beginnt Plan B: Anstatt Videospeicher wird nun Systemspeicher angefordert. Diese Funktion gibt uns eine Bitmap im normalen Speicher zurück, wenn kein Systemspeicher zur Verfügung steht. Sollten wir hier also NULL zurückgeliefert bekommen, dann gibt es wirklich ein Problem.

Da wir nun eine Bitmap mehr haben, müssen wir diese in done() auch wieder freigeben.

```
// Gibt die in init() angelegten Resourcen
// wieder frei
void done() {
    destroy_bitmap(doubleBuffer);
    if (screenAlias != NULL) {
        destroy_bitmap(screenAlias);
    }
}
```

Nachdem nun der doubleBuffer (falls möglich) im Grafikspeicher ist, gilt es jetzt, das Sprite auch in den Grafikspeicher zu bekommen. Leider gibt es keine load_video_bitmap()-Funktion, die wir für diesen Zweck benutzen könnten. Also müssen wir sie notgedrungen wohl selbst schreiben:

```
BITMAP* load_video_bitmap(const char* name) {
    BITMAP* bmp = load_bitmap(name, NULL);
    if (!bmp) {
        return NULL;
    }
    BITMAP *vid = create_video_bitmap(bmp->w, bmp->h);
    if (vid) {
        blit(bmp, vid, 0, 0, 0, 0, bmp->w, bmp->h);
        destroy_bitmap(bmp);
        return vid;
    }
    return bmp;
}
```

Nun haben wir im Idealfall alle unsere Grafiken im Videospeicher – was uns einen deutlichen Geschwindigkeitsvorteil bringt.

Allerdings bemerken wir von diesem Vorteil noch nichts, da wir bisher ja immer noch ein einziges Sprite anzeigen. Und bei nur einem Sprite kommt der Rechner (oder besser gesagt: die Grafikkarte) ja wohl nicht zum Schwitzen.

Also ändern wir das.

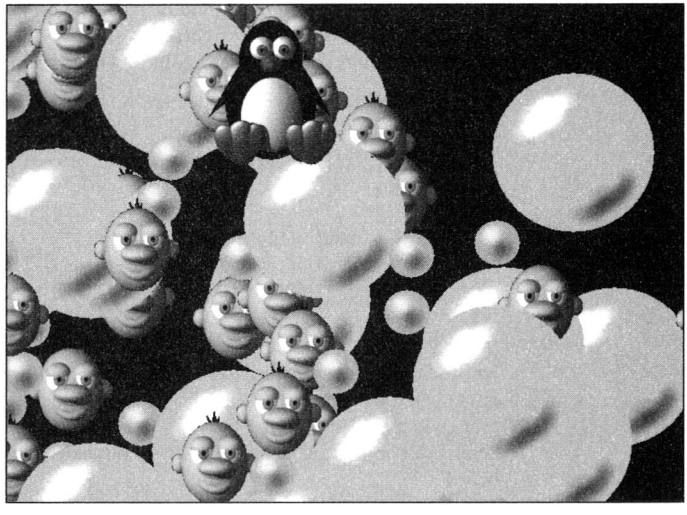


Abbildung 17.6: Screenshot der Sprite-Demo

Anstatt nur einen Pinguin über den Schirm schweben zu lassen, fügen wir noch einige andere Sprites hinzu. Zu diesem Zweck habe ich drei weitere, ebenfalls sehr simple Grafiken erstellt: zwei verschieden große Kugeln und einen Comic-Kopf.

Stören Sie sich bitte nicht an den etwas simplen Grafiken – falls sich Ihr Gefühl für Ästhetik aber dennoch beleidigt fühlt, dann ersetzen Sie einfach die Grafiken durch Bilder Ihrer Wahl. Für einen einfachen Performancetest reichen diese Bilder jedoch allemal.

Die Transformation des einsamen Pinguins zu einem gestressten Vogel in einer mit Kugeln und Köpfen gefüllten Welt verläuft in diesen einfachen Schritten:

- Wir erstellen eine Klasse, welche die Attribute der hier verwendeten Sprites kapselt. In der Tat brauchen wir hier zwei Klassen:
  - Eine für die Sprites, die sich nach links bewegen (Kugeln)
  - Eine für die Sprites, die sich nach rechts bewegen (Pinguin und Köpfe)
- Wir erstellen einen STL-Container für diese Klasse, und benutzen ihn zum Speichern der einzelnen Sprites.
- In der Logik-Schleife zeigen wir alle im Container enthaltenen Sprites an.

Eine einfache Sprite-Klasse

Ich gestehe: Ich habe gelogen. Es ist keine Sprite-Klasse, sondern »nur« eine Sprite `struct`. Der Grund dafür ist, dass ich die Attribute des Sprites bei dieser Sprite Implementierung noch nicht verstecken will. Es ist also eher eine »Struktur mit Funktionen« als eine Klasse.

Ansonsten bleibt es aber dabei. Ein Sprite in dieser Demo hat folgende Attribute.

- Bitmap – Das anzuzeigende Bild
- x, y – Die Position
- dx, dy – Die Geschwindigkeit

Darüber hinaus braucht es folgende Methoden:

- ✔ Anzeige der Grafik
- ✔ Logik-Prozedur
- ✔ Methoden, um die Attribute zu setzen

Da die Logikprozedur bei jedem Spritetyp verschieden ist, muss sie als `virtual` deklariert werden, damit sie in den abgeleiteten Klassen überladen werden kann.

```
struct Sprite {
    float x, y, dx, dy;
    BITMAP *bmp;

    Sprite(BITMAP *image) {
        bmp = image;
    }

    void draw(BITMAP *dest) {
        draw_sprite(dest, bmp, (int) x, (int) y);
    }

    void setPos(float x, float y) {
        this->x = x;
        this->y = y;
    }

    void setSpeed(float dx, float dy) {
        this->dx = dx;
        this->dy = dy;
    }

    virtual void tick() {
        x += dx;
        y += dy;

        if (x > SCREEN_W) {
            x = -bmp->w;
        }
        if (y < -bmp->h) {
            y += SCREEN_H+bmp->h;
        }
    }
};
```

Ein paar Punkte: Das Sprite alloziert keinen Speicher, auch nicht für die Bitmap. Dies erlaubt es uns die gleiche Bitmap in mehreren Sprites zu benutzen. Wenn jedes Sprite eine eigene Kopie der Bitmap hätte, würde dies eine Menge Speicher verbrauchen. Da sich die Sprites nun die Bitmap teilen, bedeutet dies aber auch, dass ein Sprite die Bitmap nicht selbst freigeben kann. Diese Aufgabe bleibt also Ihnen überlassen.

Die `tick()`-Methode ist die eigentliche Logik des Spiels. Sprites, die sich anders verhalten, müssen diese Methode überladen. Die `Bubble`-Struktur erbt von Sprite und verändert nur die `tick()`-Methode:

```
struct Bubble: public Sprite {

    Bubble(BITMAP *bmp) : Sprite(bmp) {
    }

    virtual void tick() {
        x += dx;
        y += dy;

        if (x < -bmp->w || y < -bmp->h) {
            x = rand() % SCREEN_W;
            y = SCREEN_H + bmp->h;
        }
    }
};
```

Sprites bewegen sich von links nach rechts, die Bubble-Instanzen von rechts nach links.

Diese Unterscheidung ist ziemlich willkürlich. Die Sprite-Klasse könnte ebenso die Vorzeichen von dx und dy überprüfen. Diese Lösung wäre sogar deutlich besser. Das hat man nun davon, wenn man in einem Beispiel unbedingt Vererbung und virtuelle Funktionen unterbringen möchte. Andererseits ist es eine gute Übung für Sie die Struktur so abzuändern, dass `tick()` bei der Überprüfung der Rücksetzbedingung die Bewegungsrichtung berücksichtigt.

Nachdem wir nun Klassen für diese zwei Arten von Sprites haben, bauen wir noch einen Container, um diese auch verwalten zu können.

```
typedef list<Sprite*>          SpriteList;
typedef SpriteList::iterator   SpriteIterator;
```

In diese Liste werden dann 64 Bubbles gefüllt: 32 kleine und 32 große Bläschen. Zusätzlich kommen noch 32 Sprites hinzu, die sich in die gleiche Richtung wie der Pinguin bewegen (allerdings mit einer anderen Geschwindigkeit). Ganz am Ende wird der Pinguin in diese Liste eingefügt.

```
// Die Bilder laden...
BITMAP *pingu   = load_video_bitmap("pingu.tga");
BITMAP *bubble0 = load_video_bitmap("large.tga");
BITMAP *bubble1 = load_video_bitmap("small.tga");
BITMAP *face    = load_video_bitmap("face.tga");

SpriteList spriteList;
Sprite *sprite;

const float ticks = 4.0 * 60.0;


for (int a=0; a < 32; a++) {
    sprite = new Bubble(bubble0);
    sprite->setPos(rand() % SCREEN_W,
                   rand()% SCREEN_H*2);
    sprite->setSpeed(
           (SCREEN_W / ticks * 2) / ((rand()%18)+1)*-1,
           (SCREEN_H / ticks * 2) / ((rand()%8)+1)*-1);
    spriteList.push_back(sprite);

    // Diese Bläschen sind etwas langsamer
    sprite = new Bubble(bubble1);
    sprite->setPos(rand() % SCREEN_W,
                   rand()% SCREEN_H*2);
    sprite->setSpeed(
           (SCREEN_W / ticks * 3) / ((rand()%18)+1)*-1,
           (SCREEN_H / ticks * 3) / ((rand()%8)+1)*-1);
    spriteList.push_back(sprite);

    // Und nun: Die Kopf- Sprites
    sprite = new Sprite(face);
    sprite->setPos(rand() % SCREEN_H,
                   rand() % SCREEN_H);
    sprite->setSpeed(
              SCREEN_W/ ticks / 2.0 + rand()%2,
              SCREEN_H / 8.0 / ticks *
               -1.0 - rand()%2);
    spriteList.push_back(sprite);
}
```

```
// Am Ende: Der Pinguin
sprite = new Sprite(pingu);
sprite->setPos(-sprite->bmp->w,
                (SCREEN_H - sprite->bmp->h) / 2.0);
sprite->setSpeed(SCREEN_W / ticks,
                 SCREEN_H / 4.0 / ticks * -1.0);

spriteList.push_back(sprite);
```

Jetzt haben wir alle Sprites fein säuberlich in einer Liste. Alles was es jetzt zu tun gilt, ist es die `tick()`-Methode dieser Sprites in einer Schleife aufzurufen und sie anzuzeigen.

Da wir den Pinguin zuletzt an die Liste angefügt haben, wird er nun auch als letztes gezeichnet und über allen anderen Sprites dargestellt. Die komplette Schleife, um alle Sprites anzuzeigen, ist überraschend kurz:

```
while (!key[KEY_ESC]) {
    if (timerCounter > 0) {
        while (timerCounter > 0) {
            // Hintergrund löschen
            clear_to_color(doubleBuffer,
                                    makecol(0,0, 230));
            for (SpriteIterator itor = spriteList.begin(); itor !=
spriteList.end(); ++itor) {
                sprite =*itor;
                sprite->tick();

                sprite->draw(doubleBuffer);
            }
            --timerCounter;
        }
        needsRepaint = TRUE;
    }

    if (needsRepaint) {
        needsRepaint = FALSE;
        show();
    }
}
```

Den gesamten Code finden Sie wie immer auf der beigelegten CD.

Und das war es auch schon. Spielen sie etwas mit dem Programm herum. Erhöhen Sie die Anzahl der Sprites, oder verändern Sie die `tick()`-Methode.

Animation

Die Animation bei einem Sprite funktioniert nach dem gleichen Prinzip wie ein Daumenkino. Man rendert Momentaufnahmen von einzelnen Bewegungsabläufen in rascher Abfolge auf den Schirm, und erzeugt so den Eindruck einer fließenden Bewegung. Je mehr Zwischenschritte als einzelne Bilder vorhanden sind, um so flüssiger wirkt die Bewegung.

Eine einfache Laufanimation könnte zum Beispiel aus nur drei Bildern bestehen: Rechtes Bein hinten, linkes Bein vorne; beide Beine auf gleicher Höhe; rechtes Bein vorne, linkes Bein hinten.

Drei Bilder, die in einer Animation mit 4 Schritten genutzt werden:

Das zweite Bild wird in der Animation 2mal genutzt

Abbildung 17.7: Schrittanimation

Diese Abfolge von Bildern lässt sich beliebig oft wiederholen. Es handelt sich hierbei also um einen sogenannten *Animation Loop*, eine Schleife von Bildern, die eine kontinuierliche Bewegung simulieren.

Die Schrittanimation wird auch häufig *Walk-Cycle* genannt, ein Begriff, der aus dem Zeichentrick-Genre kommt.

Eine besondere Eigenschaft der Sprite-Animation wird auch schon in diesem Beispiel deutlich: Die Anzahl der Bilder einer Animation (in diesem Beispiel also 4) muss nicht immer der Anzahl der gezeichneten (oder von einem 3D-Programm gerenderten) Einzelbildern entsprechen. Wir haben drei Einzelbilder, die zu einer Animation mit vier Bildern kombiniert werden.

Diesem Umstand müssen wir in unsere Animationsklasse Rechnung tragen. Anstatt also die Bilder direkt in der Animationsklasse zu speichern, benutzen wir eine Helferklasse, in der alle Einzelbilder gespeichert werden. Da diese Klasse die Quelle sämtlicher Einzelbilder ist, heißt sie treffendermaßen `FrameSource`. Die Animation selbst speichert nur die jeweilige `FrameSource` und den Index des Bildes. Dies erlaubt es uns, die Bilder in mehreren Animationen zu nutzen, ohne Speicherplatz zu verschwenden. Dennoch sind wir weiterhin flexibel, da jede Animation eine eigene Quelle haben kann.

```
// FrameSource in animation.h
class FrameSource {
    private:
        BITMAP *frames;
        int     count;
        int     w, h;

    public:
        FrameSource(BITMAP *bmp, int cnt);

        // bitmap wird hier nicht freigeben,
        // da sie evtl. noch an anderer Stelle
        // genutzt wird.
        ~FrameSource() {}

        // Gibt die bitmap bei explizitem Aufruf
        // frei.
        void destroy();

        // Zeigt Frame <index> an der Position (x,y) an
        void render(BITMAP *dst, int index, int x, int y);

        // Ausmasse eines Frames
        int getWidth();
```

```
        int getHeight();
};
```

Die Implementierung dieser Klasse sieht folgendermaßen aus (zu finden in `animation.cpp`):

```
FrameSource::FrameSource(BITMAP *bmp, int cnt) : frames(bmp),
count(cnt) {
        if (frames && count > 0) {
            w = frames->w / count;
            h = frames->h;
        }
    }


// Gibt die bitmap bei explizitem Aufruf
// frei.
void  FrameSource::destroy() {
    if (frames != NULL) {
        destroy_bitmap(frames);
        frames = NULL;
    }
}

// Zeigt Frame <index> an der Position (x,y) an
void FrameSource::render(BITMAP *dst, int index, int x, int y) {
    masked_blit(frames, dst, w * index, 0, x, y, w, h);
}
```

Die `FrameSource`-Klasse erwartet die einzelnen Frames in einer einzigen Bitmap. Es wird davon ausgegangen, dass alle Frames die gleichen Ausmaße haben. Dies erlaubt es uns, die Breite eines Einzelbildes aus der Breite der Bitmap und der Anzahl der Frames zu berechnen.

Zum Anzeigen wird dann der dem Frame entsprechende Ausschnitt auf die Zielbitmap geblittet (siehe Abbildung 17.8).

Eine Besonderheit hier ist die `destroy()`-Methode. Normalerweise ist es ja Aufgabe des Destruktors, den benutzten Speicher wieder freizugeben. In diesem Falle könnte dies jedoch fatale Folgen haben.

Die Bitmap mit den Einzelbildern wird mit hoher Wahrscheinlichkeit auch an anderen Stellen im Programm benötigt oder ist Teil einer Datenstruktur, die erst am Ende des Programms automatisch freigegeben werden soll (zum Beispiel weil die Bitmap in einem Allegro-Datfile gespeichert ist).

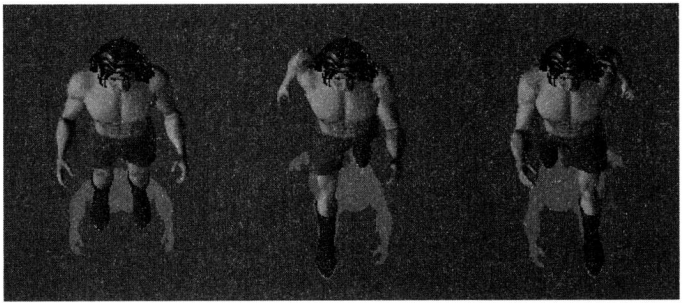


Abbildung 17.8: Die einzelnen Frames in einer `FrameSource`-kompatiblen Bitmap

Würde nun der Destruktor automatisch diese Bitmap freigeben, besteht die Gefahr, dass andere Programmabschnitte auf nicht mehr gültigen Speicher verweisen. Oder schlimmer: Der Speicherbereich könnte mehrfach freigegeben werden.

Aus diesem Grund hat die Klasse `FrameSource` eine spezielle Methode, die es uns erlaubt, den von der Bitmap belegten Speicher explizit freizugeben – oder eben auch nicht.

Neben dem Bildindex hat ein Frame noch andere Eigenschaften. Die wichtigste ist sicherlich die Zeitdauer, für die dieses Einzelbild angezeigt werden soll, bevor es weiter zum nächsten geht. Diese Dauer speichern wir wieder in der Einheit »Ticks«, wobei ein Tick der Zeitdauer entspricht, die für einen Durchlauf der Programmlogik gebraucht wird.

```
struct FrameInfo {
    public:
        int frame;
        int delay;
        int dx, dy;

        FrameInfo(int index, int time, int x, int y)
            : frame(index), delay(time), dx(x), dy(y) {
        }
};
```

Für die Feinabstimmung der Animation ist es manchmal sehr hilfreich, einzelne Bilder an einer leicht versetzten Position anzuzeigen. Damit kann man zum einen kleine Ungenauigkeiten innerhalb des Bildes aus-

gleichen, zum anderen aber auch wieder jede Menge Speicherplatz sparen.

Stellen Sie sich vor, ein Geist soll auf der Stelle schweben. Damit es nicht so langweilig wirkt, soll er sich beim »Auf-der-Stelle-Schweben« leicht nach oben und zur Seite bewegen. Ohne die Möglichkeit einen Offset anzugeben, müssten wir nun verschiedene Bilder für jede Position des Geistes anfertigen. So können wir einfach ein »Geist in der Schwebe« Bild nehmen, und dann dieses Bild mit verschiedenen Offset Werten in die Animation aufnehmen.

Die Animationsklasse selbst ist nun recht einfach zu implementieren. Ihre Hauptaufgabe ist es, einen Vector von `FrameInfo`-Strukturen zu verwalten. Dazu gehören die Anzahl der Bilder, das derzeit angezeigte Bild und Methoden, die ein problemloses Hinzufügen von neuen Frames ermöglichen.

Es ist nicht unbedingt notwendig, die Anzahl der Bilder in einer eigenen Variable zu speichern, man könnte auch den Vector fragen, wie viele Bilder er derzeit enthält. Da sich jedoch die Anzahl der Frames im Laufe des Spieles kaum ändern wird, macht es aus Performanzgründen durchaus Sinn, diese Information innerhalb der Animationsklasse zu cachen.

```
// in animation.h
typedef vector<FrameInfo>         FrameInfoVector;
typedef FrameInfoVector::iterator FrameInfoItor;

class Animation {
    private:
        FrameSource      *source;
        FrameInfoVector  frames;
        int              ticks;
        int              curFrame;
        int              count;

    public:
        Animation(FrameSource *fs);
        ~Animation();

        void addFrame(int index, int delay,
                      int dx=0, int dy = 0);

        void addFrame(FrameInfo &info);

        void tick();
```

```
        void render(BITMAP *dst, int x, int y);

        FrameSource* getFrameSource();
};

// in animation.cpp
Animation::Animation(FrameSource *src) : source(src), ticks(0),
curFrame(0), count(0) {
}

Animation::~Animation() {
}


void Animation::addFrame(int index, int delay, int dx, int dy) {
    FrameInfo info(index, delay, dx, dy);
    addFrame(info);
    ++count;
}


void Animation::addFrame(FrameInfo &info) {
    frames.push_back(info);
}

void Animation::tick() {
    if (count >0) {
        ++ticks;
        if (ticks >= frames[curFrame].delay) {
            ++curFrame;
            if (curFrame >= count) {
                curFrame = 0;
            }
            ticks = 0;
        }
    }
}

void Animation::render(BITMAP *dst, int x, int y) {
    source->render(dst,frames[curFrame].frame,
                   x + frames[curFrame].dx,
                   y + frames[curFrame].dy);
}

FrameSource* Animation::getFrameSource() {
    return source;
}
```

Im letzten Schritt müssen wir nun noch die `Sprite` Klasse so anpassen, dass sie anstatt der Bitmap eine Animation benutzt. Dies beschränkt sich jedoch auf einige kleine Änderungen in den `tick()`- und `draw()`-Methoden.

```
struct Sprite {
    float x, y, dx, dy;
    Animation *anim;

    Sprite(Animation *ani) {
        anim = ani;
    }

    void draw(BITMAP *dest) {
        anim->render(dest, (int) x, (int) y);
    }

    void setPos(float x, float y) {
        this->x = x;
        this->y = y;
    }

    void setSpeed(float dx, float dy) {
        this->dx = dx;
        this->dy = dy;
    }

    virtual void tick() {
        x += dx;
        y += dy;

        if (x > SCREEN_W) {
            x = -anim->getFrameSource()->getWidth();
        }
        if (y > SCREEN_H) {
            y = -anim->getFrameSource()->getHeight();
        }
        anim->tick();
    }
};
```

Und schon läuft unser Held über den Bildschirm.

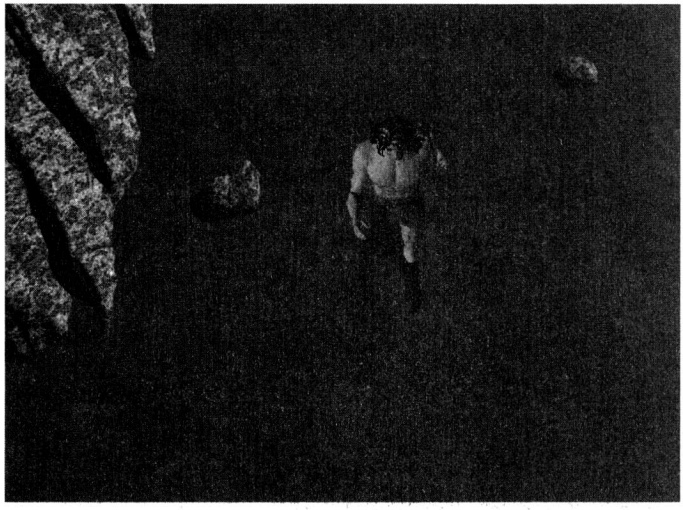


Abbildung 17.9: Der Held bewegt sich über den Schirm

Das Hauptprogramm sieht nach ein paar kleinen Anpassungen wie folgt aus:

```
int main(int argc, char** argv) {
    // Grafik modus. 640x480,
    // Logik Timer auf 60 fps
    init(640, 480, 60);

    BITMAP *frameBmp  = load_video_bitmap("hero.tga");
    BITMAP *bg        = load_video_bitmap("bg.tga");

    FrameSource frames = FrameSource(frameBmp, 3);

    Animation anim(&frames);

    anim.addFrame(0, 10);
    anim.addFrame(1, 10);
    anim.addFrame(0, 10);
    anim.addFrame(2, 10);

    Sprite sprite(&anim);
    sprite.setPos(280, 100);
```

```
    sprite.setSpeed(0, 4);

    int needsRepaint = TRUE;

    syncTimer(&timerCounter);
    while (!key[KEY_ESC]) {
        if (timerCounter > 0) {
            while (timerCounter > 0) {
                blit(bg, doubleBuffer,
                         0, 0, 0, 0, bg->w, bg->h);
                sprite.tick();
                sprite.draw(doubleBuffer);
                --timerCounter;
            }
            needsRepaint = TRUE;
        }

        if (needsRepaint) {
            needsRepaint = FALSE;
            show();
        }
    }
    destroy_bitmap(frameBmp);
    destroy_bitmap(bg);
    done();
    return 0;
} END_OF_MAIN();
```

Allerdings bewegt sich der Held immer noch etwas ungelenk über den Schirm. Bei einer aus drei Phasen bestehenden Animation ist dies auch kaum zu vermeiden. Erhöht man die Anzahl der Bilder auf sagen wir 15 dann ist die resultierende Bewegung deutlich fließender.

Und mit einer kleinen Anpassung der `main`-Routine läuft der Barbar sehr geschmeidig über den Schirm

```
for (int a=0; a < 14; a++) {
        anim.addFrame(a, 4);
}

sprite sprite(&anim);
sprite.setPos(280, 100);
sprite.setSpeed(0, 4);
```

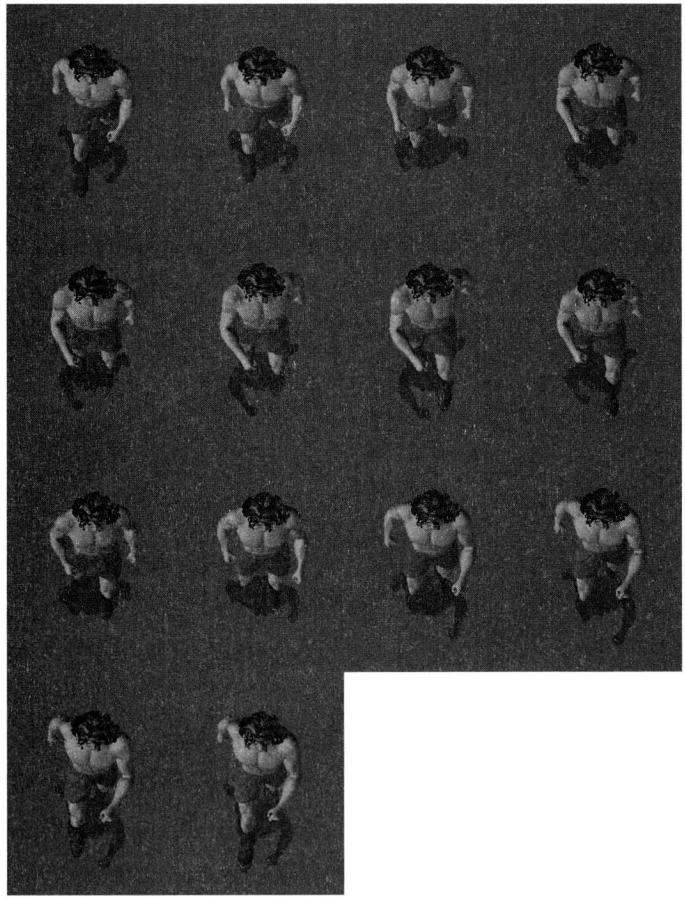


Abbildung 17.10: Walk-Cycle mit 14 Frames

Das lauffähige Programm und alle benötigten Daten finden Sie auf der CD.

18 Kollisionsabfragen

Sobald Interaktionen zwischen Objekten in einem Spiel erforderlich sind, braucht man eine Kollisionsabfrage. In diesem Kapitel gehen wir auf die Grundlagen ein und versuchen einige häufig auftretende Probleme zu lösen.

Treffen sich zwei Sprites, sagt das eine ...

Die Kollisionsabfrage ist eine der ersten Hürden, die einem das Leben so richtig schwer machen kann. Ein recht alter Witz unter Spieleentwicklern geht etwa so:

»Warum haben die Borg würfelförmige Raumschiffe? Antwort: Weil das die Kollisionsabfrage sehr viel einfacher macht.«

In der »realen Welt« ist eine Kollision ein sehr leicht zu bemerkender Effekt. Wenn uns ein Stein auf den Fuß fällt, dann müssen wir dies nicht aktiv bemerken, damit sich unser Gesicht auch entsprechend verzerrt und wir auf einem Bein durch die Gegend hüpfen können. Wenn wir aus Unachtsamkeit gegen eine Wand laufen, dann passiert es nicht, dass wir versehentlich bereits zur Hälfte in einem Stein stehen, wenn wir anhalten, oder bereits in einiger Entfernung vor der Wand zum Stillstand gezwungen werden.

In einem PC-Spiel sieht das bereits ganz anders aus. Es gibt keinerlei Hardware-Unterstützung, die es erlaubt, die Kollision zweier Grafikobjekte festzustellen.

Stellen Sie sich vor, Sie beobachten den Boxkampf zweier Gespenster. Da diese Spukgestalten keinen festen Körper haben, dringen ihre Fäuste ohne Widerstand durch den Astralkörper ihres Kontrahenten. Wenn Sie nun versuchen wollen, den Kampf zwischen den beiden Geistern real erscheinen zu lassen, würden Sie den beiden wahrscheinlich genau sagen, wie sie zu reagieren haben, wenn ein Schlag kurz davor ist, in den Gespensterkörper einzudringen.

Dinge, die im täglichen Leben von der Physik erledigt werden, müssen in genaue Regeln gefasst werden:

- Zwei Körper können nicht denselben Raum einnehmen.
- Trifft ein Körper auf den anderen, so wird ein Teil der Bewegungsenergie in Verformungsenergie umgewandelt.
- Die resultierende Bewegung resultiert aus der Bewegungsrichtung, der Masse und dem Material der Körper und deren Geschwindigkeit.

Solange Sie jedoch keinen Flipper oder eine ähnlich komplexe Simulation schreiben wollen, können Sie die physikalischen Formeln jedoch größtenteils ignorieren. Wichtig ist, dass die Ergebnisse für den Spieler schlüssig sind:

- Wenn ein Gummiball gegen eine Wand springt, dann erwartet man, dass er abprallt.
- Wenn ein Mensch gegen eine Wand springt, dann wird er in den meisten Fällen eher daran herunterrutschen (es sei denn, er hat die Fähigkeit sich von der Wand abzudrücken, um so noch höher zu springen – alles eine Frage des Spieles).

Es wird grob zwischen zwei Arten von Kollisionen unterschieden:

- Kollisionen zwischen Sprites
- Kollisionen zwischen Sprites und der Umgebung

Die leichteste Methode zur Kollisionsabfrage ist das Benutzen eines *Kollisionsrechtecks* oder auch *Bounding Rectangle* genannt.

Bounding Rectangles

Jedes Sprite hat bekanntlich eine Position und mindestens eine Bitmap. Die einfachste Methode, eine Kollision zwischen zwei Sprites festzustellen, ist also ihre Position und Größe miteinander zu vergleichen (siehe Abbildung 18.1).

Ist die Position eines Sprites durch die Werte `x` und `y` für die Position und `w` und `h` für die Ausmaße des Sprites definiert, dann kann man mit folgendem Code eine Kollision zwischen den Kollisionsrechtecken der beiden Sprites feststellen:

```
struct Rect {
    int x, y;
    int w, h;
```

```
    int isColliding(Rect &rect) {
        // Bereiche überlappen sich nicht
        // auf der X - Achse
        if (rect.x + rect.w < x || rect.x > x + w) {
            return FALSE;
        }

        // Bereiche überlappen sich nicht
        // auf der Y - Achse
        if (rect.y + rect.h < y || rect.y > y + h) {
            return FALSE;
        }
        return TRUE;
    }
};
```

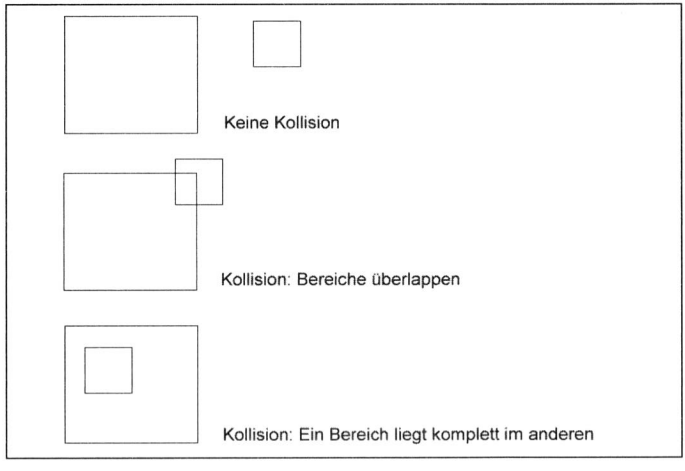


Abbildung 18.1: Mögliche Kollisionszustände

Dieser Algorithmus betrachtet die Überschneidung auf der X und Y Achse getrennt. Denn sobald auf einer dieser Achsen keine Überschneidung festgestellt werden kann, können die beiden Bereiche auch nicht übereinander liegen.

Diese Methode funktioniert sehr gut, wenn die Sprites ihr Rechteck möglichst gut ausfüllen. Gibt es allerdings leere Bereiche, dann kann es zu Problemen kommen.

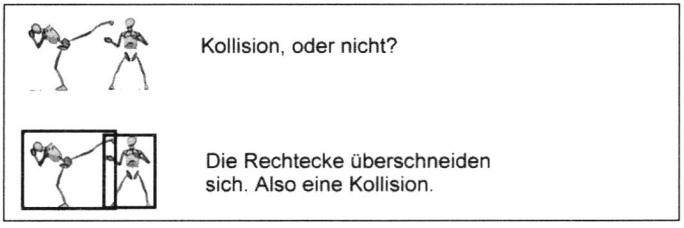


Abbildung 18.2: Falsch gemeldete Kollision

In Abbildung 18.2 sehen Sie, was passiert, wenn sich zwar die Rechtecke, nicht aber die Grafiken überlappen.

Um dieses Problem zu umgehen, können Sie die Kollisionsrechtecke unabhängig von der Größe und Position des Sprites machen.

Wenn Sie im obigen Beispiel die Größe und Position des Begrenzungsrechtecks des Verteidigers ändern, würden die Kollisionsmeldungen genauer ausfallen.

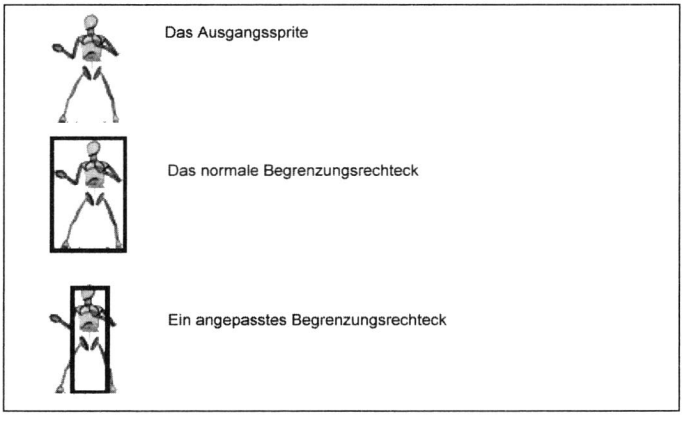


Abbildung 18.3: Angepasste Kollisionsrechtecke

Dies passt zwar beim Verteidiger recht gut, aber wie soll man das Kollisionsrechteck des Angreifers platzieren? Sowohl Kopf, Körper als auch beide Beine müssen im Rechteck enthalten sein. Dies bedeutet, dass bei einer Überschneidung im unteren Bereich (unterhalb des tretenden Beines) immer eine Kollision gemeldet werden würde.

Genauere Kollisionen

Um mit Hilfe von Kollisionsrechtecken auch bei unregelmäßig geformten Grafiken brauchbare Ergebnisse zu bekommen, kann es notwendig sein mehr als ein Rechteck zu nutzen.

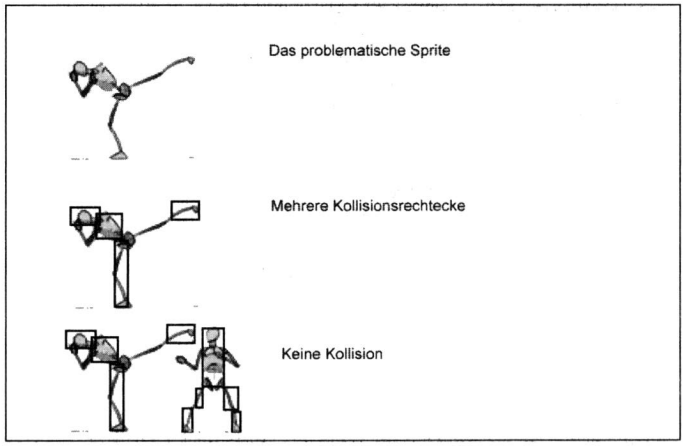


Abbildung 18.4: Mehrere Rechtecke

Ein einzelnes Sprite braucht nun also für jedes Bild der Animation, die es hat, eine Liste von Rechtecken.

```
#include <allegro.h>
#include <list>
using namespace std;

struct Rect {
    int x, y;
    int w, h;
```

```
    Rect(int x, int y, int w, int h) {
        this->x = x;
        this->y = y;
        this->w = w;
        this->h = h;
    }

    int isColliding(Rect &rect) {
        // Bereiche überlappen sich nicht
        // auf der X - Achse
        if (rect.x + rect.w < x || rect.x > x + w) {
            return FALSE;
        }

        // Bereiche überlappen sich nicht
        // auf der Y - Achse
        if (rect.y + rect.h < y || rect.y > y + h) {
            return FALSE;
        }
        return TRUE;
    }
    int isColliding(int x, int y, int x2, int y2,
                                        Rect &rect) {
        // Bereiche überlappen sich nicht
        // auf der X - Achse
        if (x2+rect.x + rect.w < x+this->x
            || x2+rect.x > this->x + x + w) {
            return FALSE;
        }

        // Bereiche überlappen sich nicht
        // auf der Y - Achse
        if (y2 + rect.y + rect.h < this->y + y
            || y2 + rect.y > this->y + y + h) {
            return FALSE;
        }
        return TRUE;
    }
};

typedef list<Rect*> RectList;
typedef RectList::iterator RectListItor;

struct Frame{
    BITMAP  *frame;
```

```
    RectList collision;

    Frame() : frame(NULL) {
    }

    void addRect(Rect &rect) {
        Rect* newRect = new Rect(rect);
        collision.push_back(newRect);
    }

    int isColliding(Frame &frame) {
        RectListItor itor;
        RectListItor destItor;
        for (itor = collision.begin(); itor !=
                            collision.end(); ++itor) {
            for (destItor = frame.collision.begin();
                    destItor != frame.collision.end();
                                        ++destItor) {
                if((*itor)->isColliding(*(*destItor))){
                    return TRUE;
                }
            }
        }
        return FALSE;
    }
    int isColliding(int x, int y, int x2, int y2,
                                    Frame &frame) {
        RectListItor itor;
        RectListItor destItor;
        for (itor = collision.begin(); itor !=
                            collision.end(); ++itor) {
            for (destItor = frame.collision.begin();
                    destItor != frame.collision.end();
                                        ++destItor) {
                if ( (*itor)->isColliding(x, y, x2, y2,
                                    *(*destItor))) {
                    return TRUE;
                }
            }
        }
        return FALSE;
    }

};
```

Die `isColliding()`-Routine des Frames vergleicht jedes Rechteck in der eigenen `collision`-Liste mit jedem Rechteck in der `collision`-Liste des übergebenen Frames, bis eine Kollision festgestellt wurde.

Beachten Sie, dass die `isColliding()`-Methoden nun auch `x, y, x2, y2` Parameter übergeben bekommen. Diese geben die Position des Frames auf den Bildschirm an (Die `Rects` in der Liste sind relativ zur oberen linken Ecke der Grafik.).

Wir können die Kollisionsabfrage noch etwas beschleunigen, indem wir zuerst prüfen, ob sich die beiden Bilder überlagern. Falls dies nicht der Fall sein sollte, ist es auch nicht notwendig die `Rect`-Listen zu vergleichen.

```
int isColliding(int x, int y, int x2, int y2, Frame &frame) {
    Rect self(x, y, this->frame->w, this->frame->h);
    Rect other(x2, y2, frame.frame->w, frame.frame->h);

    if (!self.isColliding(other)) {
        return FALSE;
    }
   //... siehe oben..
}
```

Im Hauptprogramm müssen Sie nun noch die Kollisionsabfragen durchführen. Das komplette Programm sieht dann so aus:

```
#include <allegro.h>
#include <list>

#include "util.h"


using namespace std;

int WHITE;

struct Rect {
    int x, y;
    int w, h;

    Rect(int x, int y, int w, int h) {
        this->x = x;
        this->y = y;
        this->w = w;
        this->h = h;
    }
```

```
    int isColliding(Rect &rect) {
        // Bereiche überlappen sich nicht auf der X - Achse
        if (rect.x + rect.w < x || rect.x > x + w) {
            return FALSE;
        }

        // Bereiche überlappen sich nicht auf der Y - Achse
        if (rect.y + rect.h < y || rect.y > y + h) {
            return FALSE;
        }
        return TRUE;
    }

    int isColliding(int x, int y, int x2, int y2, Rect &rect) {
        // Bereiche überlappen sich nicht auf der X - Achse
        if (x2+rect.x + rect.w < x+this->x || x2+rect.x > this->x + x
      + w) {
            return FALSE;
        }

        // Bereiche überlappen sich nicht auf der Y - Achse
        if (y2 + rect.y + rect.h < this->y + y || y2 + rect.y > this-
>y + y + h) {
            return FALSE;
        }
        return TRUE;
    }
};

typedef list<Rect*> RectList;
typedef RectList::iterator RectListItor;

struct Frame{
    BITMAP  *frame;
    int     shouldDestroy;
    RectList collision;

    Frame() : frame(NULL),shouldDestroy(0) {
    }

    Frame(BITMAP *bmp) : frame(bmp),shouldDestroy(0) {
    }

    Frame(const char *str):shouldDestroy(1) {
```

```
    frame = load_bitmap(str, NULL);

    int    argc = 0;
    char** argv = get_config_argv(str, "rect0", &argc);
    char   buffer[256];
    int    cur   = 0;


    while (argv != NULL) {
        if (argc == 4) {
            int x = atoi(argv[0]);
            int y = atoi(argv[1]);
            int w = atoi(argv[2]);
            int h = atoi(argv[3]);

            collision.push_back(new Rect(x,y,w,h));
        }
        ++cur;
        sprintf(buffer, "rect%i", cur);
        argv = get_config_argv(str, buffer, &argc);
    }
}

Frame(const Frame &frame) {
    copy(frame);
}

~Frame() {
    if (frame && shouldDestroy) {
        destroy_bitmap(frame);
        frame = NULL;
    }
    RectListItor itor = collision.begin();
    for(; itor != collision.end(); ++itor) {
        delete *itor;
    }
}



Frame& operator=(const Frame& frame) {
    if (this != &frame) {
        copy(frame);
    }
    return *this;
}
```

```
    void addRect(Rect &rect) {
        Rect* newRect = new Rect(rect);
        collision.push_back(newRect);
    }

    int isColliding(int x, int y, int x2, int y2, Frame &frame) {
        Rect self(x, y, this->frame->w, this->frame->h);
        Rect other(x2, y2, frame.frame->w, frame.frame->h);

        if (!self.isColliding(other)) {
            return FALSE;
        }
        RectListItor itor;
        RectListItor destItor;
        for (itor = collision.begin(); itor != collision.end();
      ++itor) {
            for (destItor = frame.collision.begin(); destItor !=
            frame.collision.end(); ++destItor) {
                if ( (*itor)->isColliding(x, y, x2, y2,
               *(*destItor))) {
                    return TRUE;
                }
            }
        }
        return FALSE;
    }
    private:
        void copy(const Frame& rhs) {
            this->shouldDestroy = 0;
            this->frame         = rhs.frame;
            RectListItor itor = ((Frame*)(&rhs))->collision.begin();
            for(; itor != rhs.collision.end(); ++itor) {
                collision.push_back(new Rect(*(*itor)));
            }
        }
};

struct Sprite {
    float x, y;
    float dx, dy;
    Frame *frame;

    int   ignore;

    Sprite(Frame *f) : frame(f), ignore(20) {
        x = rnd(SCREEN_W - f->frame->w);
```

```
        y = rnd(SCREEN_H - f->frame->h);

        dx = ((float) rnd(200)) / 20.0;
        dy = ((float) rnd(200)) / 20.0;
    }

    void tick() {
        x+=dx;
        y+=dy;

        if (x < 0 || x > SCREEN_W - frame->frame->w) {
            dx *= -1;
        }
        if (y < 0 || y > SCREEN_H - frame->frame->h) {
            dy *= -1;
        }
        x = MID(0, x , SCREEN_W - frame->frame->w);
        y = MID(0, y , SCREEN_H - frame->frame->h);
    }

    void show() {
        draw_sprite(doubleBuffer, frame->frame, (int)x, (int)y);
    }

    int isColliding(Sprite &spr) {
        if (ignore > 0) {
            --ignore;
            //return 0;
        }
        int result = frame->isColliding((int) x,(int) y, (int)spr.x,
        (int)spr.y, *spr.frame);
        if (result) {
            ignore = 5;
        }
        return result;
    }
};

typedef list<Sprite*> SpriteList;
typedef SpriteList::iterator SpriteIterator;

int main(int, char**){

    // Höhe, Breite, Bilder pro Sekunde
    init(640, 480, 60);
    WHITE = makecol(255,255,255);
```

```
set_config_file("collision.ini");

SpriteList sprites;

Frame  *ball = new Frame("ball.tga");

for (int a=0; a < 10; a++) {
    sprites.push_back(new Sprite(ball));
}

int needsRefresh = FALSE;

timerCounter = 0;
syncTimer(&timerCounter);

while (!key[KEY_ESC]) {
    if (timerCounter) {
        while (timerCounter >0) {
            --timerCounter;
            clear_to_color(doubleBuffer, WHITE);
            for (SpriteIterator itor = sprites.begin(); itor !=
            sprites.end(); ++itor) {
                (*itor)->tick();
                for (SpriteIterator itor2 = sprites.begin();
                itor2 != sprites.end(); ++itor2) {
                    if ((*itor != *itor2) && (*itor)-
                    >isColliding(**itor2)) {
                        (*itor)->dx *= -1;
                        (*itor)->dy *= -1;
                        (*itor2)->dx *= -1;
                        (*itor2)->dy *= -1;
                        break;
                    }
                }
                (*itor)->show();
            }
        }
        needsRefresh = 1;


        //draw_sprite(doubleBuffer, ball->frame, 100, 100);
    }
    if (needsRefresh) {
        show();
        needsRefresh = 0;
    }
```

```
    }

    delete ball;

    done();
    return 0;
} END_OF_MAIN()
```

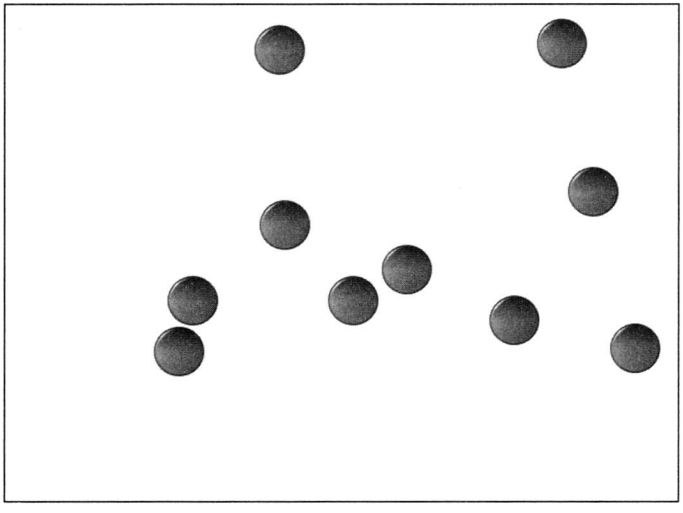


Abbildung 18.5: Zehn Bälle mit Kollisionsabfrage

Bitte beachten Sie auch das `ignore`-Attribut der Sprite-Klasse. In der `isColliding()`-Methode wird das `ignore` Attribut nach jeder Kollision auf einen fixen Wert gesetzt. Auf diese Weise können Sie ein »Festhaken« einzelner Sprites verhindern. Dies kann nötig werden, falls Sie sehr viele Sprites auf dem Schirm haben. In diesem Fall kann es zum Beispiel vorkommen, dass sich zwei Sprites schon zu Beginn überlappen und aufgrund ähnlicher Richtungen auch nicht auseinander kommen, ohne eine weitere Kollision auszulösen.

19 Scrolling

Scrolling erlaubt es Ihnen, nahtlos von einem Bereich Ihres Spielfeldes zu einem anderen zu gelangen. Der Spieler betrachtet die Welt durch einen sich sanft bewegenden Ausschnitt.

Die ersten Computerspiele hatten alle nur ein Bildschirmlayout. Spielte man eine Videospielvariante von Tennis, dann gab es eine Mittellinie, zwei Seitenlinien, den Ball und die Schläger der beiden Spieler. Das war es auch schon. Bewegt man sich durch ein Labyrinth, dann hatte man immer alles im Überblick.

Die technische Entwicklung ermöglichte es aber recht bald, mehrere Spielfelder in die Spiele zu integrieren. Hatte man dann in einem Labyrinthspiel alle Gegner abgeschossen, so kam man (nach einer kurzen Pause) in den nächsten Raum. Trotzdem blieb jedes Spielfeld für sich immer überschaubar und eine in sich geschlossene Einheit. Im Jahre 1980 wurden die ersten Spiele veröffentlicht, bei denen das Spielfeld größer war als der sichtbare Bildschirmausschnitt.

Bewegte man sich bei diesen Spielen nahe an einen der Ränder heran, dann bewegte sich der Bildschirmausschnitt zusammen mit der Spielfigur. Heutzutage ist dies nichts Besonderes mehr, aber damals war es eine ziemliche Sensation.

Durch *Scrolling* war die Größe des Bildschirms auf einmal nicht mehr das Limit. Scrolling ist eigentlich ein Kunstwort, zusammengesetzt aus Screen und Rolling – Bildschirmrollen.

Was bei der Hardware im Jahre 1980 eine Meisterleistung war, ist für heutige Verhältnisse selbstverständlich. Aber wie implementiert man Scrolling?

Im einfachsten Fall laden Sie eine Bitmap, die größer ist als der eigentliche Schirm, und zeigen dann immer einen etwas anderen Ausschnitt von diesem Bild auf dem Schirm an.

Die `blit()`-Methode erlaubt es Ihnen ja, den oberen linken Punkt und die Größe des anzuzeigenden Bildausschnitts anzugeben. Wenn Ihr Spielfeld komplett als Bitmap im Speicher steht, dann können Sie also einen scrollenden Hintergrund mit einer einzigen Befehlszeile abhandeln:

```
blit(levelBg, doubleBuffer, x, y, 0, 0, SCREEN_W, SCREEN_H);
```

Wobei (x,y) die linke obere Ecke angibt und davon ausgegangen wird, dass der komplette Schirm zur Anzeige des Spielfeldes genutzt wird.

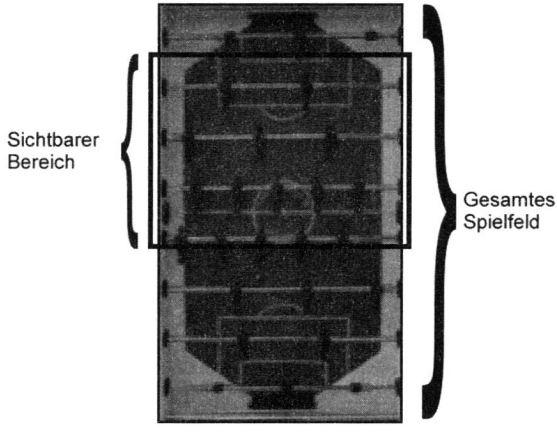


Abbildung 19.1: Konzept beim Scrolling

Behalten Sie bei dieser Lösung aber bitte im Kopf, dass sie wirklich speicheraufwendig ist. Nehmen wir an, Sie wollen in einem Hüpfspiel einen Level haben, der acht Schirme breit und drei Schirme hoch ist. Das wären folglich 8 * 3 = 24 Schirme. Ein Bild in 640 x 480 Pixel bei 16 Bit-Farbtiefe belegt 640 * 480 * 2 = 614400 Bytes, also rund 600 Kilobyte. Bei 24 Schirmen wären dies bereits über 14 MB. Zwar ist dies bei Hauptspeichergrößen von derzeit 256 MB bis zum Gigabyte-Bereich kein KO-Kriterium, diese Methode nicht zu benutzen, aber es gibt einen schon zu denken. Insbesondere wenn Sie Ihr Spiel über das Internet vermarkten möchten, ist die Downloadgröße ein wichtiger Faktor.

In einigen Fällen sind vorberechnete Hintergründe jedoch die beste Möglichkeit. Dies würde es Ihnen zum Beispiel erlauben, die Szenen mit sehr viel Detail im voraus zu berechnen. Wenn Ihr Spiel in einem Spukhaus spielt, dann könnten Sie jeden Raum einzeln in einem 3D-Rendering-Programm erstellen und dann mit dem Programm verbinden. Grosse Räume könnten sich dann über 3 x 3 Schirme erstrecken, kleine nur einen Ausschnitt des Bildschirms füllen und lange Korridore sich lang in eine Richtung erstrecken.

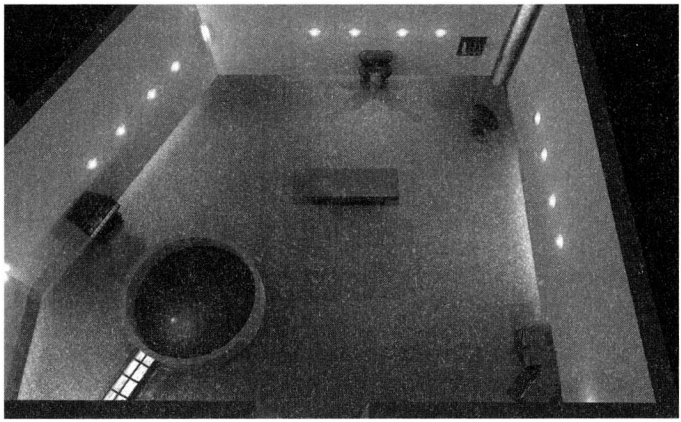


Abbildung 19.2: Eine »Location« in einem verlassenen Labor

Auch können Sie auf diese Weise auch den Blickwinkel der Kamera von Raum zu Raum ändern, was Ihrem Spiel etwas von einem Film gibt. Sie können die harten Schnitte zwischen den einzelnen Kameraeinstellungen benutzen, um die Grafiken für den nächsten Raum nachzuladen.

Es gibt natürlich auch Methoden um, größere Karten speicherplatzsparend abzuspeichern. Allerdings geht dies meist auf Kosten der Details in den Levels. Wenn Sie allerdings komplette Städte anstatt eines einzelnen Hauses als Schauplatz für Ihr Spiel wählen, dann ist es ratsam, auf vorberechnete Bilder zu verzichten, falls Sie vorhaben, das Spiel über das Internet zu vertreiben.

Virtuelle Kamera

In den meisten Spielen folgt der Bildschirmausschnitt dem Sprite des Spielers. Auf dem ersten Blick mag es aus diesem Grund sinnvoll erscheinen, das Scrolling von der Position der Spielfigur abhängig zu machen. Diese Methode hat einige Vorteile. Zum einen ist sie sehr einfach zu implementieren, da sich die obere linke Ecke meist über zwei einfache Berechnungen finden lässt:

```
upperLeft.x = player.x - HALF_SCREEN_WIDTH;
upperLeft.y = player.y - HALF_SCREEN_HEIGHT:
```

Da sich HALF_SCREEN_WIDTH und HALF_SCREEN_HEIGHT im Voraus berechnen lassen, lässt sich der Bildschirmausschnitt auf sehr einfache Weise festlegen.

Der Nachteil dieser Methode ist jedoch die Inflexibilität. Wenn man zum Beispiel für eine Zwischensequenz das andere Ende eines Levels zeigen möchte oder auch die Art und Weise ändern möchte, durch die der sichtbare Bereich festgelegt wird, dann stoßen Sie hier an Grenzen. In den meisten Fällen wird dann etwas getrickst, um das gewünschte Ergebnis zu erzielen (so könnte eine Variable angeben, welche Methode derzeit benutzt wird um den sichtbaren Ausschnitt zu bestimmen). Allerdings macht man so auch den großen Vorteil, nämlich die Schlichtheit wieder zunichte.

Oder stellen Sie sich vor, Sie wollen mehreren Spielern das gleichzeitige Spielen an nur einem Rechner ermöglichen. Dies würde bedeuten, dass Sie zwei (oder mehr) Hauptcharaktere haben, die sich möglicherweise recht weit voneinander entfernen.

Welches der von den Usern gesteuerten Sprites beeinflusst nun den Bildausschnitt? Wie können wir mit diesem System ein *Splitscreen*, also das Darstellen unterschiedlicher Ansichten auf dem gleichen Schirm, ermöglichen?

Die Antwort auf diese Frage lautet: gar nicht. Wir müssen uns also etwas Neues einfallen lassen.

Wenn das Spiel ein Film wäre, dann wäre während der Dreharbeiten immer eine Kamera auf den Hauptdarsteller gerichtet. Die meiste Zeit würde sie wahrscheinlich versuchen den Darsteller im Zentrum zu halten, aber in bestimmten Situationen, zum Beispiel wenn eine Unterhaltung von mehreren Personen stattfindet, kann es sinnvoll sein die Kamera etwas anders auszurichten.

Wenn es nun notwendig wird, einen anderen Teil des Levels anzuzeigen, kann entweder die Kamera an diesen Ort springen, oder es wird auf eine andere Kamera umgeschaltet, die sich an dieser Position befindet.

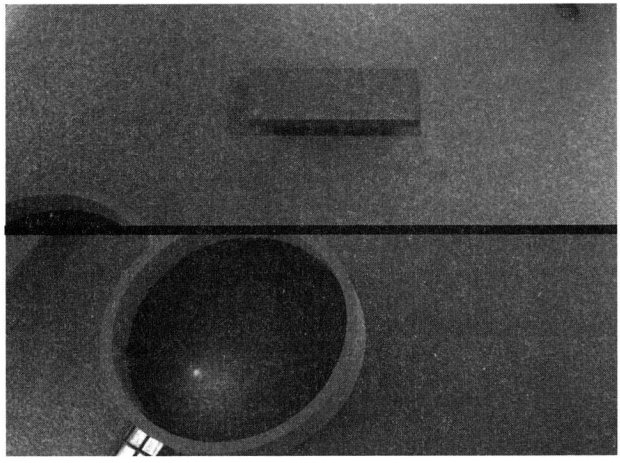


Abbildung 19.3: Splitscreen in Aktion

Wenn man mehrere gleichzeitige Kameras hat, dann kann man auch sehr einfach Splitscreen-Effekte implementieren. In einem weiteren Schritt kann man dann nicht nur die Eigenschaften der Kamera (Größe des Bildes und der erfasste Ausschnitt) sondern auch das Verhalten der Kamera in ein eigenes Objekt auslagern.

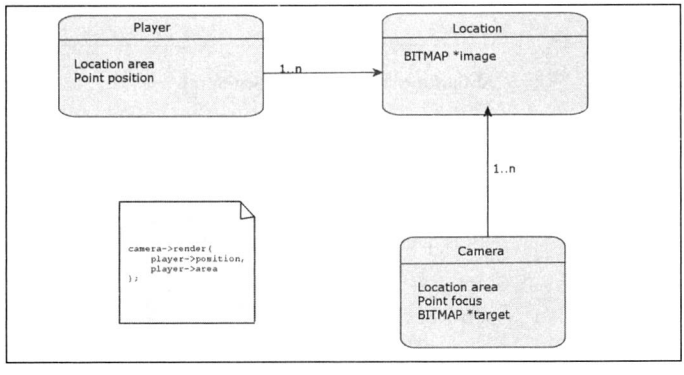


Abbildung 19.4: Zusammenhang von Player, Location und Camera

Ein Location-Objekt besteht in erster Linie aus dem geladenen Bild, das dem Level entspricht. Allerdings könnte man hier auch Kollisionsinformationen speichern und festlegen, wo sich Ein- und Ausgänge befinden.

Das Player-Objekt speichert die Location, an der sich der Spieler derzeit befindet, und auch die position innerhalb dieser Location.

Um nun etwas auf dem Schirm anzuzeigen, ruft man das Camera-Objekt mit den benötigten Informationen auf. Zu diesen Informationen gehören die Location und der focus der Kamera. Der focus ist die Position, welche die Kamera bestmöglichst darstellen soll. Dies könnte man erreichen, indem man den focus innerhalb des Ausgabebereiches zentriert. Der Ausgabebereich ist ein Zeiger auf eine BITMAP.

Um nun einen Splitscreen-Effekt zu erhalten, teilt man den doubleBuffer in zwei Sub-Bitmaps auf und erstellt zwei Kameras, die je einen der beiden Hälften als target zugewiesen bekommen.

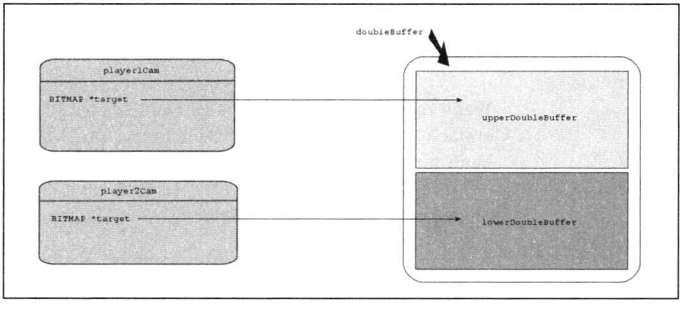


Abbildung 19.5: Splitscreen-Setup

Bisher wurden die Objekte meist als Strukturen implementiert. Dieses mal nehmen wir Klassen und stellen sicher, dass nur über die Zugriffsmethoden auf die Attribute zugegriffen werden kann. Die Unterschiede im eigentlichen Code sind eher gering, allerdings erfordern die Klassen etwas mehr Schreibaufwand.

```
class Location {

private:
    BITMAP *image;
public:
```

```
    Location (const char* filename) {
        image = NULL;

        BITMAP *img = load_bitmap(filename, NULL);
        if (img) {
            if (is_video_bitmap(doubleBuffer)) {
                image = create_video_bitmap(
                                img->w, img->h);
                if (image) {
                    blit(img, image, 0, 0, 0, 0,
                         img->w, img->h);
                    destroy_bitmap(img);
                }
            }
        }
        if (!image) {
            image = img;
        }
    }
    ~Location () {
        if (image) {
            destroy_bitmap(image);
        }
    }

    BITMAP *getImage() {
        return image;
    }

    int getWidth() {
        if (image) {
            return image->w;
        }
        return 0;
    }
    int getHeight() {
        if (image) {
            return image->h;
        }
        return 0;
    }
};
```

Neben der Verwaltung des Hintergrundbildes (durch `setImage()` und `getImage()`) kann der Benutzer auch die Ausmaße des geladenen Bildes mit `getWidth()` und `getHeight()` abfragen. Alle Methoden dieser Klasse

sind recht einfach gehalten, nur der Konstruktor enthält etwas Extralogik. Er überprüft, ob der doubleBuffer im Speicher der Grafikkarte gehalten wird, und legt das geladene Bild im gleichen Speicher an, sofern möglich.

Auch die Camera Klasse hat nur eine komplexere Methode.

```
class Camera {
private:
    BITMAP *target;
public:
    Camera() : target(NULL) {
    }

    Camera(BITMAP *target) {
        setTarget(target);
    }

    ~Camera() {
        // Nichts zu tun.
        // Die targets werden außerhalb verwaltet
    }

    void setTarget(BITMAP *target) {
        this->target = target;
    }

    BITMAP *getTarget() {
        return target;
    }

    void render(Location *area, int x, int y) {
        if (!target) {
            return;
        }
        int startX = x - target->w / 2;
        int startY = y - target->h / 2;

        int destX = 0;
        int destY = 0;
        int w = target->w;
        int h = target->h;

        if (startX < 0 || startY < 0
           || startX + target->w > area->getWidth()
           || startY + target->h > area->getHeight()) {
```

```
            clear(target);
            destX = startX < 0 ? startX *-1 : 0;
            destY = startY < 0 ? startY *-1 : 0;

            w = MIN(target->w,
                    area->getWidth()  - startX);
            h = MIN(target->h,
                    area->getHeight() - startY);
        }

        blit(area->getImage(), target,
             startX, startY, destX, destY, w, h);
    }
};
```

Die `render()`-Methode legt fest, wie die Gegend um den darzustellenden Punkt angezeigt wird. In diesem Fall wird der Fokuspunkt immer im Zentrum dargestellt. Werden dadurch Bereiche sichtbar, die außerhalb der `Location` liegen, wird erst das `target` gelöscht und dann der sichtbare Bereich an der entsprechenden Position angezeigt.

Damit haben wir zwei der drei Klassen, die wir für ein Scrolling inklusive Splitscreen-Effekt brauchen, bereits abgearbeitet. Die dritte Klasse, der `Player`, braucht allerdings ein paar Extraeigenschaften. In der Übersicht hat er ja in erster Linie die Position und die `Location` verwaltet. Nun erhält er auch Methoden für Benutzereingaben und speichert auch die für ihn verwendete `Camera`.

```
class Player {

private:
    Location *area;
    int       x, y;

    int ctrl[4];
    int deltaX[4];
    int deltaY[4];

    Camera   *cam;

public:

    Player() : area(0), x(0), y(0) {
        for (int a=0; a < 4; a++) {
            ctrl[a]  = 0;
        }
```

```
    deltaX[0] =  0; deltaY[0] = -1;
    deltaX[1] =  0; deltaY[1] =  1;
    deltaX[2] = -1; deltaY[2] =  0;
    deltaX[3] =  1; deltaY[3] =  0;
}

Player(Location* area) {
    this->area = area;
    setX(area->getWidth()/2);
    setY(area->getHeight()/2);

    for (int a=0; a < 4; a++) {
        ctrl[a]  = 0;
    }
    deltaX[0] =  0; deltaY[0] = -1;
    deltaX[1] =  0; deltaY[1] =  1;
    deltaX[2] = -1; deltaY[2] =  0;
    deltaX[3] =  1; deltaY[3] =  0;
}

int getX() {
    return x;
}
void setX(int x) {
    this->x = x;
}

int getY() {
    return y;
}
void setY(int y) {
    this->y = y;
}

void setPosition(int x, int y) {
    setX(x);
    setY(y);
}

void setArea(Location *area) {
    this->area = area;
}
Location *getArea() {
    return area;
}
```

```
    void setCamera(Camera *cam) {
        this->cam = cam;
    }

    Camera *getCamera() {
        return cam;
    }

    void render() {
        getCamera()->render(getArea(), getX(), getY());
    }

    void setControls(int up, int down, int left, int right) {
        ctrl[0] = up;
        ctrl[1] = down;
        ctrl[2] = left;
        ctrl[3] = right;
    }

    void reactOnUserInput() {
        for (int a=0; a < 4; a++) {
            if (key[ctrl[a]]) {
                setX(getX() + deltaX[a]);
                setY(getY() + deltaY[a]);
            }
        }
    }
};
```

Für die Eingabeverwaltung wird ein ähnliches Prinzip verwendet wie es in Kapitel 12 beschrieben wurde. Fügt man nun alles zusammen, dann erhält man mit vergleichsweise geringem Aufwand ein Programm, in dem man zwei Ansichten unabhängig voneinander scrollen lassen kann.

```
int main(int, char**){
    const int COUNT_PLAYERS = 2;

    // Höhe, Breite, Bilder pro Sekunde
    init(640, 480, 60);

    Location *area   = new Location("room.tga");

    Player *player[COUNT_PLAYERS];
    BITMAP *target[COUNT_PLAYERS];

    target[0] = create_sub_bitmap(doubleBuffer,
                                  0, 0, SCREEN_W/2, SCREEN_H);
```

```
    target[1] = create_sub_bitmap(doubleBuffer,
                  SCREEN_W/2, 0, SCREEN_W/2, SCREEN_H);
    for (int a=0; a < COUNT_PLAYERS; a++) {
        player[a] = new Player(area);
        player[a]->setCamera(new Camera(target[a]));
    }
    player[0]->setControls(
                KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT);
    player[1]->setControls(
                KEY_W, KEY_S, KEY_A, KEY_D);

    int       needsRefresh = FALSE;

    timerCounter = 0;
    syncTimer(&timerCounter);

    while (!key[KEY_ESC]) {
        if (timerCounter) {
            while (timerCounter) {
                --timerCounter;

                for (int a=0; a < COUNT_PLAYERS; ++a) {
                    player[a]->render();
                    player[a]->reactOnUserInput();
                }
            }
            needsRefresh = 1;
        }
        if (needsRefresh) {
            show();
            needsRefresh = 0;
        }
    }

    delete area;
    for (int a=0; a < COUNT_PLAYERS; a++) {
        delete player[a]->getCamera();
        delete player[a];
        delete target[a];

    }
    done();
    return 0;
} END_OF_MAIN()
```

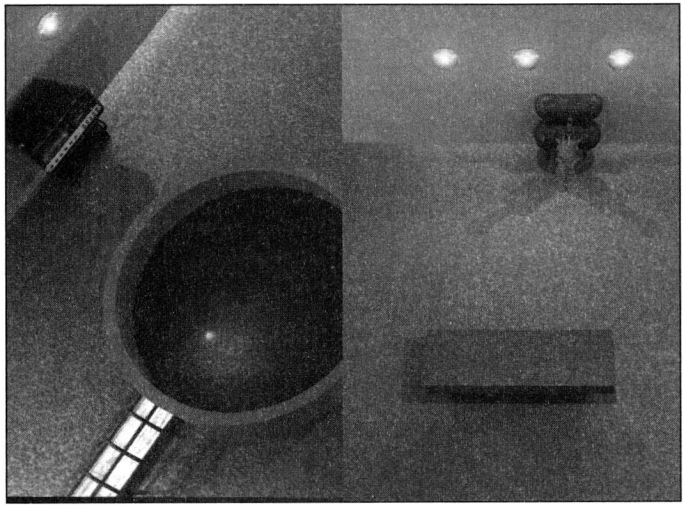


Abbildung 19.6: Die Beispielanwendung in Aktion

Übungen

- ✔ Ändern Sie `Player` so ab, dass das virtuelle Joypad aus Kapitel 12 für Eingaben benutzt wird.
- ✔ Ändern Sie das Programm so ab, dass mit den Tasten 1, 2, 3 und 4 die Anzahl der sichtbaren Ausschnitte gesetzt wird.
- ✔ Benutzen Sie den Kollisionscode aus Kapitel 18, um Kollisionsrechtecke in einer `Location` zu platzieren. Verknüpfen Sie diese Rechtecke mit einer weiteren `Location`, so dass ein neuer Bereich geladen wird, sobald die im `Player` gespeicherte Position innerhalb eines der Rechtecke liegt.

Parallax Scrolling

Wenn man einen Tiefeneffekt erzielen möchte, ist Parallax Scrolling eine der Möglichkeiten. Eine Parallaxe ist die »scheinbare Verschiebung eines Objektes durch Änderung des Betrachtungsstandpunktes.« Klingt kom-

pliziert, ist es aber nicht. Schließen Sie ein Auge, und zeigen Sie mit Ihrem Zeigefinger auf einen beliebigen Punkt in Ihrer Umgebung. Und jetzt öffnen Sie das geschlossene Auge und schließen Sie das bisher offene. Ihr Zeigefinger deutet nun nicht mehr auf das ursprüngliche Objekt. Je weiter das Objekt entfernt war, desto größer ist die Abweichung.

Unser Gehirn hat dies gelernt, und wir nutzen so diese Abweichung, um Entfernungen einschätzen zu können. Dies kann man sich zu Nutze machen. Wenn wir in einem seitwärts scrollendem Spiel, zum Beispiel in einem Ballerspiel oder einem Plattformspiel, die weiter hinten liegenden Elemente langsamer bewegen als die, die näher am Betrachter sind, erhalten wir einen Tiefeneffekt.

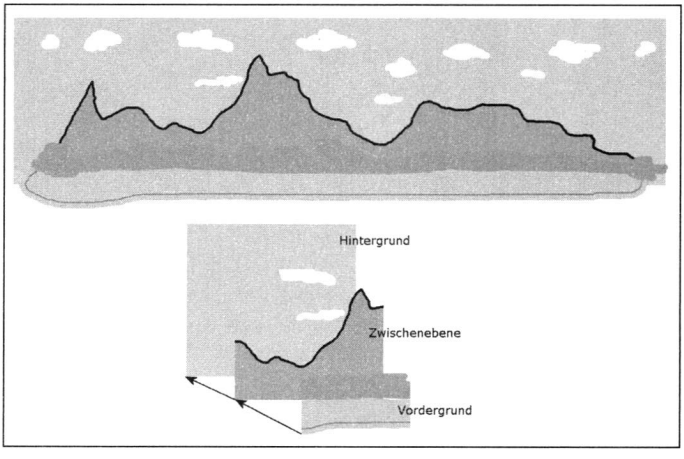


Abbildung 19.7: Parallax Scrolling

Abbildung 19.7 zeigt die typische Anordnung von Vordergrundebene (auf der sich der Spieler bewegt), der Zwischenebene und dem Hintergrund.

Mit diesen drei Ebenen können Sie bereits einen sehr guten Tiefeneffekt erzielen. Eine weitere Steigerung wäre noch eine weitere Ebene vor dem Vordergrund zu platzieren, die teilweise auch das Spielersprite verdecken könnte. Dies sorgt nochmals für eine Steigerung der Tiefenwirkung, allerdings kann das Verdecken von Sprites schnell zu Frust führen, wenn der Spieler seine eigene Figur, heranstürmende Gegner oder einen Ab-

grund nicht rechtzeitig sehen kann. Verwenden Sie also eine solche Ebene immer mit Bedacht.

Derzeit gehen wir immer noch davon aus, dass unsere Level aus vorgefertigten Bitmaps bestehen (dies ändert sich in Teil 3 dieses Buches). Und nun haben wir nicht nur eine Bitmap, sondern sogar drei. Dadurch wächst der benötigte Speicher deutlich an.

Da sich die Ebenen unterschiedlich schnell bewegen, brauchen sie auch nicht die gleiche Größe zu haben. Wenn der Vordergrund mit vier Pixel pro Frame scrollt, die Zwischenebene mit zwei Pixel und der Hintergrund mit einem, folgt daraus, dass die Zwischenebene nur halb so breit sein muss wie der Vordergrund. Bei der Hintergrundebene reicht sogar nur ein Viertel der Breite aus.

Wenn der Vordergrund fünf Schirme breit ist, dann ergibt sich folgende Situation:

	Anzahl Schirme	Pixel
Vordergrund	5	640 * 5 = 3200
Zwischenebene	2.5	640 * 2.5 = 1600
Hintergrund	1.25	640 * 1.25 = 800

Tabelle 19.1: Berechnung der Ebenenbreiten

Aber nicht nur die Breite, auch in der Höhe kann man einiges an Raum sparen. Dies spart zum einen Speicherplatz, zum anderen wird dadurch auch die Anzeige beschleunigt.

Wenn Sie Abbildung 19.7 noch einmal betrachten, dann fällt Ihnen sicher auf, dass die Berge nie ganz bis zum oberen Rand des Bildschirmes gehen. Auch verdeckt der Fuß der Berge immer ein gutes Stück des Hintergrunds. Der Vordergrund bleibt meist im unteren Drittel der Berge.

Wenn wir dies in der Planung unserer Parallax Routinen berücksichtigen, können wir einiges an Zeit und Speicherplatz sparen.

Wenn wir von drei Ebenen ausgehen, dann können wir ein Array nehmen, um alle Daten zu speichern. Diese Lösung ist zwar nicht so flexibel wie die Nutzung eines Vektors oder einer Liste, aber in diesem Fall reicht ein Array aus. Falls Sie sich entscheiden die Anzahl der Ebenen zu ver-

ringern, dann können Sie den entsprechenden Code sehr schnell anpassen, um für die neue Anzahl von Ebenen gewappnet zu sein. Sogar eine Anpassung an eine dynamische Anzahl von Ebenen ist leicht möglich. Sie haben also auf Sourcecode-Ebene eine hohe Flexibilität und die Implementierung geht auch noch schneller von der Hand.

Ausgehend von dem bestehenden Code passen wir erst die `Location`-Klasse an. Im ersten Schritt definieren wir über ein anonymes `enum` die nötigen Klassenkonstanten.

```
class Location {
public:
    enum {
        BACK   = 0,
        MIDDLE = 1,
        FRONT  = 2,
        COUNT  = 3
    };
   // ... etc ...
}
```

Dadurch können wir nun über `Location::COUNT` von außen auf die Anzahl der Ebenen zugreifen. Und auch die individuellen Ebenen können nun über einen symbolischen Namen angesprochen werden.

Da wir nun mehrere Ebenen haben, brauchen wir auch mehr als nur eine Bitmap. Und für jede Ebene kommt noch der Y-Offset und der Faktor in Vergleich zur Vordergrundebene dazu.

```
private:
    BITMAP *image[COUNT];
    int     yOffset[COUNT];
    float   factor[COUNT];
```

Diese Informationen (Bitmapname und Offset) können sehr bequem in einem Allegro Config File gespeichert werden. Der Dateiname dieser INI-Datei wird der `load()`-Methode der `Location` übergeben.

```
void load(const char* filename) {
    char *section[] = { "bg", "mg", "fg" };

    // Es kann nur ein config file aktiv
    // sein. Also muss erst der derzeitige
    // Zustand gespeichert werden
    push_config_state();
    set_config_file(filename);
```

```
    for (int a=0; a < COUNT; ++a) {
        if (image[a] != NULL) {
            destroy_bitmap(image[a]);
            image[a] = NULL;
        }

        image[a]   = loadVideoBitmap(
                         get_config_string(section[a],
                         "img", NULL));
        yOffset[a] = get_config_int(section[a],
                         "y", 0);
    }
    if (image[FRONT]->w <= doubleBuffer->w) {
        // kein Scrolling
        factor[FRONT] = factor[MIDDLE]
                      = factor[BACK]
                      = 0;
    } else {
        factor[BACK]
          = (float)(image[BACK]->w   -doubleBuffer->w)
          / (float)(image[FRONT]->w  -doubleBuffer->w);
        factor[MIDDLE]
          = (float)(image[MIDDLE]->w -doubleBuffer->w)
          / (float)(image[FRONT]->w  -doubleBuffer->w);
        factor[FRONT]  =  1.0;
    }

    pop_config_state();
}
```

Neben dem Einlesen der Daten und Laden der Bilder berechnet die Methode auch die Verhältnisse der Ebenen zur Vordergrundebene. Beachten Sie hierbei, dass Sie von der Gesamtbreite der Bilder die Breite des Bildschirms abziehen müssen, um die Breite des tatsächlich scrollenden Bereichs zu berechnen.

Die `INI`-Dateien haben folgendes Format:

```
[bg]
img=bg.tga
[mg]
img=mg.tga
y=98
[fg]
img=fg.tga
y=312
```

Das Laden (und die Konvertierung in eine Video-Bitmap) ist in eine eigene Methode ausgelagert worden. Diese Methode, `loadVideoBitmap()`, ist als `private` deklariert.

```
BITMAP *loadVideoBitmap(const char *filename) {

    if (!filename) {
        return NULL;
    }

    BITMAP *image = NULL;

    BITMAP *img = load_bitmap(filename, NULL);
    if (img) {
        if (is_video_bitmap(doubleBuffer)) {
            image = create_video_bitmap(img->w, img->h);
            if (image) {
                blit(img, image, 0, 0, 0, 0, img->w, img->h);
                destroy_bitmap(img);
            }
        }
    }
    if (!image) {
        return img;
    }
    return image;
}
```

Die Zugriffsmethoden werden um einen Parameter (mit Default-Wert) erweitert, damit der Zugriff auf alle Ebenen möglich wird.

```
BITMAP *getImage(int layer=0) {
    return image[layer];
}

int getWidth(int layer=0) {
    if (image[layer]) {
        return image[layer]->w;
    }
    return 0;
}
int getHeight(int layer=0) {
    if (image) {
        return image[layer]->h;
    }
    return 0;
```

```
}
int getYOffset(int layer=0) {
    return yOffset[layer];
}

float getFactor(int layer=0) {
    return factor[layer];
}
```

Die nächste größere Änderung muss an der `Camera`-Klasse vorgenommen werden. Sobald die `render()`-Methode angepasst wurde, funktioniert das restliche Programm wie gehabt.

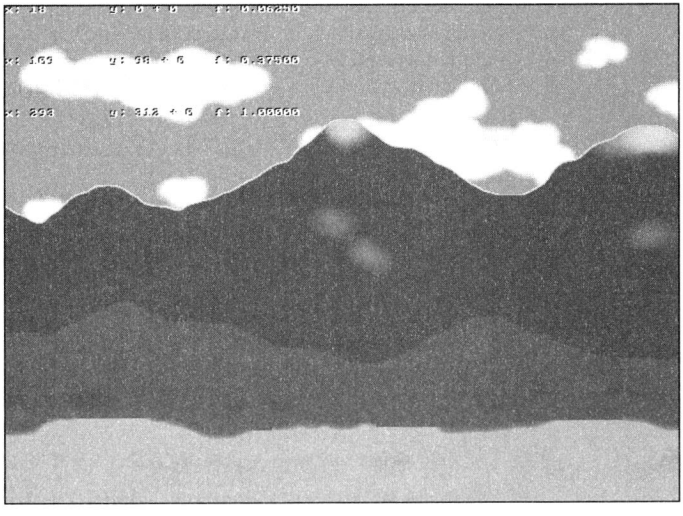


Abbildung 19.8: Die Parallax-Demo im Fullscreenmodus

Innerhalb der `render()`-Methode ist der erste Schritt wieder das Anpassen der übergebenen Parameter an die `target`-Bitmap und die Ebenengröße.

```
int halfWidth  = target->w/2;
int halfHeight = target->h/2;
```

```
x = MIN(x, area->getWidth(Location::FRONT)  - halfWidth);
y = MIN(y, area->getHeight(Location::FRONT) - halfHeight);

int x0 = MAX(x - halfWidth,  0);
int y0 = MAX(y - halfHeight, 0);

// Die Offset Werte im INI File gehen von
// einer Fullscreen Application aus
// Der Y-Offset muss an die Target Größe
// angepasst werden.
int ofsY = target->h -SCREEN_H;
```

Der nächste Schritt ist das Anzeigen der einzelnen Ebenen. Wir beginnen mit der Hintergrundebene (`Location::BACK`) und blitten dann die mittlere und schließlich die Vordergrundebene. Von diesen dreien kann nur die Hintergrundebene »normal« geblittet werden, bei den anderen Ebenen muss auf transparente Bereiche geachtet werden.

```
blit(area->getImage(Location::BACK), target,
        (int)(x0 * area->getFactor(Location::BACK)),
        0,0,
        area->getYOffset(Location::BACK) + ofsY ,
        target->w, area->getHeight(Location::BACK));
masked_blit(area->getImage(Location::MIDDLE), target,
        (int)(x0 * area->getFactor(Location::MIDDLE)),
        0,0,
        area->getYOffset(Location::MIDDLE) + ofsY,
        target->w, area->getHeight(Location::MIDDLE));
masked_blit(area->getImage(Location::FRONT), target,
        x0,
        0,0,
        area->getYOffset(Location::FRONT) + ofsY ,
        target->w, area->getHeight(Location::FRONT));
```

Wenn das Makro DEBUG gesetzt ist, werden Zusatzinformationen ausgegeben, die bei der Fehlersuche behilflich sein können. Dazu gehören die Offsets der einzelnen Ebenen, die Scrollpositionen und die Faktoren.

```
#ifdef DEBUG
for (int a=0; a < Location::COUNT; ++a) {
    textprintf(target, font, 0, a*50, 0,
            "x: %i", (int)(x0 * area->getFactor(a)));
    textprintf(target, font, 1, a*50+1,
              makecol(255,255,255),
             "x: %i", (int)(x0 * area->getFactor(a)));
```

```
    textprintf(target, font, 100, a*50, 0,
              "y: %i + %i", area->getYOffset(a), ofsY);
    textprintf(target, font, 101, a*50+1,
              makecol(255,255,255),
              "y: %i + %i", area->getYOffset(a), ofsY);

    textprintf(target, font, 200, a*50, 0,
                "f: %2.5f ", area->getFactor(a));
    textprintf(target, font, 201, a*50+1,
               makecol(255,255,255),
               "f: %2.5f ", area->getFactor(a));
}
#endif
```

Am eigentlichen Demo-Programm wurde nur wenig geändert. Anstatt des Namens der Bilddatei wird nun der Name der INI-Datei an den Constructor des `Location`-Objektes übergeben. Die restlichen Änderungen sind nur kosmetischer Natur. So werden nun die Splitscreen `target`-Bereiche innerhalb einer Schleife berechnet.

```
int main(int, char**){
    const int COUNT_PLAYERS = 2;

    // Höhe, Breite, Bilder pro Sekunde
    init(640, 480, 60);

    Location *area   = new Location("parallax.ini");
    Player *player[COUNT_PLAYERS];
    BITMAP *target[COUNT_PLAYERS];

    for (int a=0; a < COUNT_PLAYERS; a++) {
        target[a] = create_sub_bitmap(doubleBuffer,
                    0, SCREEN_H/COUNT_PLAYERS*a,
                    SCREEN_W, SCREEN_H/COUNT_PLAYERS);
        player[a] = new Player(area);
        player[a]->setCamera(new Camera(target[a]));
        player[a]->setY(SCREEN_H/2);
    }
    player[0]->setControls(
                KEY_UP, KEY_DOWN, KEY_LEFT, KEY_RIGHT);
    if (COUNT_PLAYERS > 1) {
        player[1]->setControls(
                KEY_W, KEY_S, KEY_A, KEY_D);
    }
```

```
    int       needsRefresh = FALSE;

    timerCounter = 0;
    syncTimer(&timerCounter);

    while (!key[KEY_ESC]) {
        if (timerCounter) {
            while (timerCounter) {
                --timerCounter;

                for (int a=0; a < COUNT_PLAYERS; ++a) {
                    player[a]->render();
                    player[a]->reactOnUserInput();
                }
            }
            needsRefresh = 1;
        }
        if (needsRefresh) {
            show();
            needsRefresh = 0;
        }
    }

    delete area;
    for (int a=0; a < COUNT_PLAYERS; a++) {
        delete player[a]->getCamera();
        delete player[a];
        delete target[a];

    }
    done();
    return 0;
} END_OF_MAIN()
```

Das Ergebnis ist ein parallax-scrollendes Gebirge in einer Splitscreen-Ansicht (siehe Abbildung 19.9).

Aufgrund des geringen Ausschnitts geht allerdings ein Großteil des Effektes verloren. Eine Alternative wäre es, die Daten zuerst in voller Größe in einen weiteren Zwischenspeicher zu blitten und diesen dann zu skalieren.

Allerdings geht dabei einiges an Geschwindigkeit verloren.

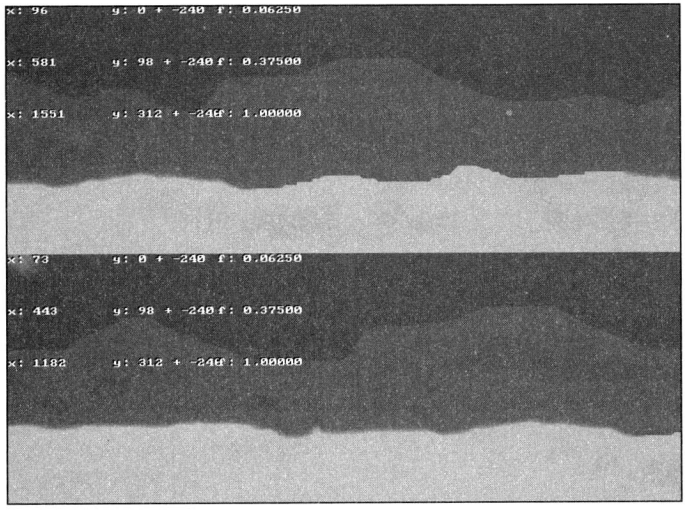


Abbildung 19.9: Splitscreen und Parallax Scrolling

Abschließende Bemerkungen

Riesige Grafiken zu laden und damit im Hauptspeicher zu jonglieren wird von vielen Spieleprogrammierern (insbesondere abseits der kommerziellen Studios) als »schlechter Stil« angesehen. Die gleichen Programmierer sprechen aber mit glänzenden Augen von der gerenderten Grafik des aktuellen Top-Spieletitels. Lassen Sie sich also davon nicht entmutigen. Wenn Sie eine Spielidee haben, die sich gut mit einer solchen Lösung umsetzen lässt, dann nutzen Sie sie.

Falls Sie vorhaben das fertige Spiel dann über das Internet zu verteilen und Ihnen die Downloadgröße Sorgen macht, dann können Sie immer noch auf Zusatzbibliotheken für Allegro zurückgreifen, die Ihnen das Laden von hochkomprimierten Dateien wie PNG und JPEG erlauben.

Und natürlich gehen wir in den folgenden Kapiteln auch auf eine Lösung ein, die nicht auf vorberechneten Bitmaps basiert. Nutzen Sie die Lösung, die Ihnen bei Ihrem Projekt am sinnvollsten erscheint.

C++-Puristen verdammen die Verwendung von `structs`. Anhänger von C können mit der STL nichts anfangen. Lassen Sie sich davon nicht verrückt machen. Nutzen Sie, was Ihnen sinnvoll erscheint. Aber seien Sie sich auch immer über die Vor- und Nachteile der jeweiligen Lösung bewusst.

20 Soundeffekte

Die Geräuschkulisse eines Spieles beeinflusst den Spieler mehr als Sie vielleicht glauben mögen. Genau wie bei Filmen spielen Soundeffekte auch bei Spielen eine sehr große Rolle.

Ein Spiel mit schlechten Soundeffekten wird keinen großen Erfolg haben. Allerdings kann eine gute Geräuschuntermalung in Verbindung mit einfachen Grafiken zu sehr ansprechenden Ergebnissen führen.

Obwohl wir die meisten Geräusche gar nicht bewusst wahrnehmen, so fällt uns doch das Fehlen derselben sehr schnell auf. Wenn Sie im Kino sitzen und einen Film sehen, dann können Sie in den meisten Fällen die Augen schließen und immer noch sehr viel von der Handlung und auch von der Stimmung der einzelnen Szenen mitbekommen.

Wenn Sie jedoch eine rührende Szene einmal ohne Hintergrundmusik gehört haben oder eine epische Schlacht einmal in Stille betrachten, dann sind diese Szenen auf einmal weit weniger bewegend.

Achten Sie einmal beim Betrachten eines Films besonders auf die Hintergrundmusik und spezielle Soundeffekte. Wann fängt man an, den Herzschlag der Titelheldin zu hören? Mit welchem Geräusch wird der überraschende Sprung der Katze aus dem Schatten untermalt? Und, was hat Sie mehr erschreckt: Das plötzliche Auftauchen des Stubentigers oder der Einsatz des Orchesters in diesem Moment?

Ein Spiel, welches Geräusche nahezu perfekt einsetzt, ist »Eternal Darkness« für den Nintendo Gamecube. In diesem Horror-Adventure sind Sie ständig von leisen Stimmen umgeben, die Ihnen Dinge zuflüstern. Ein Schockeffekt jagt den anderen und die Geräusche und die unheimliche Musik können Ihnen einen Schauer über die Haut jagen.

In diesem Kapitel werde ich Ihnen zeigen, wie Sie eigene Sounds erstellen können. Zwar gibt es auch eine Vielzahl von CDs mit Sound-Collections, aber in den meisten Fällen ist genau das Geräusch nicht dabei, das Sie brauchen.

Mit einem Mikrofon bewaffnet, ist es allerdings kein Hexenwerk die Sounds, die benötigt werden, selbst zu erstellen. Für die ganz ausgefallenen Geräusche kann man auf Software-Synthesizer zurückgreifen.

Geräusche selbst aufnehmen

Um Geräusche selbst aufzunehmen, brauchen Sie auf alle Fälle ein gutes Mikrofon. Es muss keine Studioqualität haben, aber es sollte die Geräusche auch nicht verfremden. Sie können versuchen Testberichte in Magazinen und im Internet zu durchforsten, aber in letzter Zeit sind nur sehr wenige Mikrofone für den Hobbybereich getestet worden. Und der Preis für ein Mikrofon in Studioqualität kann einem Tränen in die Augen treiben.

In den meisten Fällen wird Ihnen also nichts übrigbleiben als entweder das Mikrofon zu benutzen, das Sie bereits besitzen (egal wie gut oder schlecht es auch sein mag), oder im Laden zu testen, welches Mikrofon nun das beste Ergebnis liefert.

Idealerweise haben Sie nicht nur ein Mikro, sondern auch ein tragbares Aufnahmegerät. Hier ist natürlich digitales Equipment ideal, aber der tragbare Kassettenrecorder kann ein guter Ersatz sein.

- ✔ Regel 1 ist: Seien Sie kreativ.
  Brauchen Sie Degen- und Schwertgeklapper, dann reicht ein Gang in die Küche. Zwei Gabeln, die gegeneinander schlagen, haben schon in vielen Mantel- und Degenfilmen dafür gesorgt, dass die Kämpfe lebendiger wirken. Experimentieren Sie mit Aufprallwinkeln und Flächen herum, bis Sie eine Klangfarbe haben, die Ihnen zusagt.
  Es gibt eine Anekdote von einem Geräuschemacher einer bekannten Science-Fiction-Serie, der für die passende Geräuschuntermalung für das Innere eines fremden Raumschiffs die Geräusche seines Magens nach dem Mittagessen aufgenommen hat. Er hat dann noch die Abspielgeschwindigkeit verringert, damit das Geräusch tiefer klingt.

- ✔ Regel 2: Hören Sie genauer hin.
  Schließen Sie hin und wieder mal die Augen und konzentrieren Sie sich komplett auf die Sie umgebende Geräuschkulisse. Wie viele verschiedene Geräusche können Sie hören? Wie viele Geräusche sind eigentlich gleich, aber unterscheiden sich in Nuancen? Können Sie die Synchronstimmen im Fernsehen den anderen Schauspielern zuordnen, die vom gleichen Sprecher nachvertont werden?

Kampfgeräusche

Für Schwerterklirren eignen sich Gabeln ideal. Die Zinken schwingen hervorragend mit und erzeugen so das bekannte Geräusch aus den Abenteuerfilmen.

Um den Aufschlag einer Waffe auf einem Schild zu simulieren, schlagen Sie mit der Gabel kurz auf die Außenseite einer Schöpfkelle. Wenn Sie die Gabel dabei nachwippen lassen, können Sie so auch das Geräusch des Schwertschmiedens nachempfinden.

Für unbewaffnete Angriffe brauchen Sie ein Handtuch oder eine Jeanshose als Utensil. Drücken Sie den Stoff zusammen und ziehen Sie ihn wieder möglichst rasch auseinander. Damit können Sie das Geräusch eines Karate- oder Kung-Fu-Anzuges bei einem Schlag oder Tritt simulieren.

Das Geräusch beim Aufprall einer Faust können Sie wie folgt erzeugen: Füllen Sie einen mittelgroßen Stoffbeutel mit Reis oder Erbsen. Schlagen Sie nun mit dem Handballen auf diesen Beutel. Einen Körpertreffer simulieren Sie, indem Sie zwei dieser Beutel gegeneinander schlagen.

Für einen Schuss brauchen Sie ein langes Plastiklineal und eine Holz- oder Plastikschachtel (zum Beispiel von einer Videohülle). Spannen Sie das Lineal und lassen Sie es auf die Schachtel schnalzen.

Bewegung

Füllen Sie einen kleinen Sack mit Mehl. Wenn Sie diesen rhythmisch zusammendrücken erhalten Sie ein Geräusch wie von Schritten im Schnee.

Pferdegeklapper kann durch das Aneinanderschlagen zweier Kokosnüsse simuliert werden. Gegebenenfalls erhalten Sie bessere Ergebnisse, wenn Sie die Kokosnüsse vorher mit Stoff umwickeln.

Wenn sich der Held mühsam mit seinem Schwert einen Weg durch den Dschungel oder das Unterholz bahnt, dann brauchen Sie nicht mehr als eine Gabel (als altbewährter Messer-/Degen-/Schwertersatz) und ein Bündel mit Stroh.

Schritte nehmen Sie am besten direkt auf. Suchen Sie sich passende Schuhe und laufen Sie auf einem passenden Untergrund (Steinplatten oder Kiesel).

Umgebungsgeräusche

Ein knisterndes Feuer können Sie mit Zellophanfolie (zum Beispiel in Form von Geschenkpapier) simulieren. Zerknistern Sie die Folie und variieren Sie dabei den Abstand zum Mikrofon.

Erbsen, die Sie auf ein Backblech oder auf eine mit Klarsichtfolie bespannte Schüssel prasseln, sind ein hervorragender Ersatz für Regen.

Wind können Sie recht einfach durch kontrolliertes Ausatmen bei weit geöffneten Mund nachahmen. Ändern Sie gegebenenfalls die Tonhöhe oder die Abspielgeschwindigkeit in Ihrem Soundeditor.

Eine knarrende und quietschende Tür kann als »sich öffnende Schatztruhe« zu neuen Ehren kommen.

Um zersplitterndes Glas aufzunehmen, brauchen Sie ein Trinkglas und etwas Klarsichtfolie. Spannen Sie die Folie über die Öffnung des Glases. Nehmen Sie nun einen kleinen Schlüssel und drücken Sie ihn durch die Folie. Das entsehende Geräusch klingt wie zersplitterndes Glas (auf diese Weise kann man sich auch in einem Restaurant die Aufmerksamkeit des Kellners verschaffen – aber seien Sie auf misstrauische Blicke vorbereitet).

Geräusche verändern

Ihnen wird bei den Aufnahmeversuchen aufgefallen sein, dass keine zwei Versuche genau gleich klingen. Sie können diese Vielfältigkeit in Ihr Programm übernehmen, indem Sie zwei (oder noch mehr) Varianten jedes Geräuschs aufnehmen und dann weitere Varianten erzeugen, indem Sie die Abspielgeschwindigkeit minimal nach oben oder unten regulieren. Bei 5 – 10 % Abweichung ist das eigentliche Geräusch noch gut erkennbar, die Änderung in der Tonhöhe und Abspieldauer erzeugt aber den Eindruck eines abgeänderten Sounds.

```
#include <allegro.h>
#include "util.h"

void playSound(SAMPLE *sample, int vol) {
    int freq= rnd(100) + 950;

    vol *= 5+(rnd(5)+1);
    vol /= 10;
    play_sample(sample, vol, 128, freq, 0);
}
```

```
int main(int, char**){
    // Höhe, Breite, Bilder pro Sekunde
    init(400, 200, 60);

    const char* text = "<esc> beendet / "
                       "<space> spielt den sound";

    int    needsRefresh = FALSE;
    int    spaceDown    = 0;
    BITMAP *bg        = load_bitmap("bg.tga", NULL);
    SAMPLE *sample[3];

    char buffer[200];
    for (int a=0; a < 3; a++) {
        sprintf(buffer, "sword%i.wav", a);
        sample[a]= load_sample(buffer);
    }

    int snd = 0;

    timerCounter = 0;
    syncTimer(&timerCounter);

    while (!key[KEY_ESC]) {
        if (timerCounter) {
            while (timerCounter) {
                --timerCounter;
            }
            blit(bg, doubleBuffer,0,0,0,0,bg->w,bg->h);
            textprintf_centre(doubleBuffer, font,
                              doubleBuffer->w/2,
                              doubleBuffer->h - 20,
                              makecol(255,255,255),
                              text);
            needsRefresh = 1;
        }
        if (key[KEY_SPACE]) {
            if (!spaceDown) {
                spaceDown = 1;
                snd = rnd(3);
                playSound(sample[snd], 200);
            }
        } else {
            spaceDown = 0;
        }
```

```
        if (needsRefresh) {
            show();
            needsRefresh = 0;
        }
    }

    destroy_bitmap(bg);
    for (int a=0; a < 3; a++) {
        destroy_sample(sample[a]);
    }

    done();
    return 0;
} END_OF_MAIN()
```

Dieses Beispiel spielt drei unterschiedliche Schwertkampfgeräusche ab. Diese Geräusche werden auch noch zufällig in der Frequenz und Lautstärke variiert.

Abbildung 20.1: Das Beispiel-Programm

Generell möchte ich Ihnen empfehlen, Ihre Sounds auch mal deutlich schneller und langsamer und auch rückwärts anzuhören. Auf diese Weise lassen Sich zum Teil ganz neue Anwendungsmöglichkeiten finden.

Sounds/Musik selbst erzeugen

Wenn Sie Soundfiles nur abspielen wollen, dann können Sie die SAMPLE-Struktur ohne jedes Problem als eine Blackbox betrachten. Sie laden Ihre Soundfiles mit load_sample() und geben sie später mit destroy_sample() wieder frei. Aber was, wenn Sie nun ein SAMPLE dynamisch erzeugen wollen?

Ein SAMPLE hat folgende Struktur:

```
typedef struct SAMPLE
{
   int bits;                  /* 8 oder 16          */
   int stereo;                /* stereo?            */
   int freq;                  /* sample Frequenz    */
   int priority;              /* 0-255              */
   unsigned long len;         /* Länge (in samples) */
   unsigned long loop_start;  /* loop Start         */
   unsigned long loop_end;    /* loop Ende          */
   unsigned long param;       /* Reserviert         */
   void *data;                /* sample data        */
} SAMPLE;
```

Bei einem 8-Bit-Sample stehen 256 verschiedene Werte für die Darstellung eines bestimmten Sounds da. Wenn Sie Musikstücke und/oder Geräusche haben, bei denen es zu feinen Variationen kommt, verlieren Sie eventuell etwas an Genauigkeit.

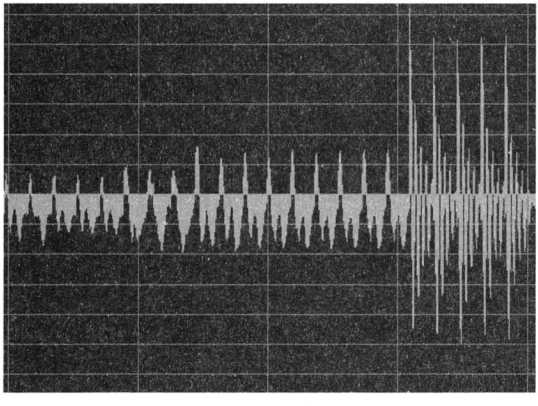


Abbildung 20.2: Sampledaten grafisch dargestellt

Die Frequenz gibt an, wie oft pro Sekunde eine Schallwelle abgetastet wird. Bei niedriger Frequenz kann es dann vorkommen, dass die Form der Schallwelle so verfremdet wird, dass sich die Klangqualität hörbar verschlechtert. Stellen Sie sich vor, der Schall hat in einem bestimmten Beispiel eine Sinusform. Wenn Sie von dieser Sinuskurve jeden zweiten Punkt weglassen (die Abtastfrequenz halbieren), dann wird aus der Kurve ein deutlich eckigeres Gebilde. Mit der Zeit nähert sich dann die Sinuskurve einer Dreieckskurve an.

Bei einem Stereosample verdoppelt sich die Anzahl der Daten, da nun die Informationen getrennt für den linken und rechten Kanal gespeichert werden.

Die Daten (`data`) sind ein Array von Integern. Bei einem 8-Bit-Sample sind die Daten vorzeichenbehaftet (-128 bis +127), bei 16 Bit werden die Daten ohne Vorzeichen gespeichert. In diesem Fall befindet sich die Mittellinie bei 0x8000 hexadezimal oder 32768 dezimal.

Nehmen wir an, Sie wollen ein 16-Bit-Sample, mono bei 44.1kHz erzeugen. Sie sind nur ein Aufruf einer Allegro-Funktion davon entfernt.

```
Sample *sample = create_sample(16, 0, 44100, len);
```

Sie übergeben der Funktion die Anzahl der Bits (8, 16), ob Sie ein Stereosample erzeugen wollen (0 = mono, 1= stereo), die Samplefrequenz (häufig benutzt werden 11025, 22050 und 44100) und schließlich die Anzahl der Samples. Pro Sekunde benötigen Sie so viele Samples wie es von der Frequenz angegeben wird.

```
int freq = 44100;
int t      =  60; //1 Minute = 60 Sekunden
SAMPLE* sample = create_sample(16, 0, freq, t* freq);
```

Ist der Rückgabewert ungleich `NULL`, dann können Sie ab diesem Zeitpunkt beliebig auf den `data` Zeiger `SAMPLE` Struktur zugreifen.

Die Rechteckschwingung lässt sich am einfachsten erzeugen:

```
void createRectWave(SAMPLE* sample, int freq) {
    memset(sample->data, 0x8000, sample->len *2 );

    if (freq == 0) {
        return;
    }
    int dist  = sample->freq / freq;
    int dist2 = dist/2;
    int count = sample->len / dist +1;
```

```
    for (int a=0; a < count; a++) {
        for (int b=0; b < dist; b++) {
            if (a*dist+b >= sample->len) {
                return;
            }
            if (b < dist2) {
                ((short*)sample->data)[a*dist+b] = 0xffff;
            } else {
                ((short*)sample->data)[a*dist+b] = 0x0000;
            }
        }
    }
}
```

Das Ergebnis dieser Funktion ist eine rechteckförmige Schwingung. Je nach übergebener Frequenz können Sie damit tiefe (geringe Frequenz) oder hohe Töne (hohe Frequenz) erzeugen.

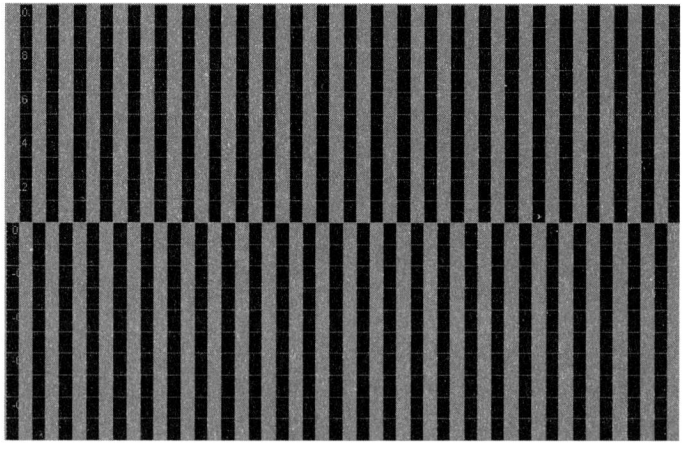


Abbildung 20.3: Eine Rechteckschwingung

Das Beispielprogramm sound2 erzeugt immer höhere Frequenzen. Sie können das Programm mit Esc verlassen oder mit Leer das aktuelle Sample abspeichern.

Beim Abspeichern kann man es sich zu Nutze machen, dass das WAV Format vor den eigentlichen Sampledaten nur einen kurzen Informationsblock abspeichert.

```
int export_sample(AL_CONST char *filename, SAMPLE *spl) {
    int bps = spl->bits/8 * ((spl->stereo) ? 2 : 1);
    int len = spl->len * bps;
    int i;
    signed short s;
    PACKFILE *f;

    f = pack_fopen(filename, F_WRITE);

    if (f) {
        // HEADER Start
        //--------------
        /* RIFF header */
        pack_fputs("RIFF", f);
        /* size of RIFF chunk */
        pack_iputl(36+len, f);
        /* WAV definition */
        pack_fputs("WAVE", f);
        /* format chunk */
        pack_fputs("fmt ", f);
        /* size of format chunk */
        pack_iputl(16, f);
        /* PCM data */
        pack_iputw(1, f);
        /* mono/stereo data */
        pack_iputw((spl->stereo) ? 2 : 1, f);
        /* sample frequency */
        pack_iputl(spl->freq, f);
        /* avg. bytes per sec */
        pack_iputl(spl->freq*bps, f);
        /* block alignment */
        pack_iputw(bps, f);
        /* bits per sample */
        pack_iputw(spl->bits, f);

        // HEADER Ende
        //--------------
        /* data chunk */
        pack_fputs("data", f);
        /* actual data length */
        pack_iputl(len, f);
        if (spl->bits == 8) {
```

```
            /* write the data */
            pack_fwrite(spl->data, len, f);
        } else {
            for (i=0; i < (int)spl->len * ((spl->stereo) ? 2 : 1);
           i++) {
                s = ((signed short *)spl->data)[i];
                pack_iputw(s^0x8000, f);
            }
        }

        pack_fclose(f);
    }
    return (errno == 0);
}
```

Diese Funktion sieht auf den ersten Blick etwas kompliziert aus. Dies liegt vor allem daran, dass die Allegro-File-Funktionen benutzt werden. Das eigentliche Abspeichern der Sounddaten wird durch die unspektakuläre Schleife gegen Ende der Funktion erledigt.

Im Hauptprogramm wird in der Logikschleife überprüft, ob das Sample noch immer abgespielt wird. Sobald festgestellt wird, dass das Sample beendet ist, wird die Frequenz erhöht, die Sampledaten neu berechnet, und schließlich das Sample wieder abgespielt.

Hier ist das Hauptprogramm:

```
int main(int, char**){
    // Höhe, Breite, Bilder pro Sekunde
    init(400, 200, 60);

    char *text = "<esc> beendet / "
                 "<space> speichert das sample";

    int    needsRefresh = FALSE;
    int    spaceDown    = 0;
    BITMAP *bg          = load_bitmap("bg.tga", NULL);
    SAMPLE *sample;

    sample = create_sample(16, 0, 44100, 44100);

    int snd   = 0;
    int freq  = 0;
    createRectWave(sample, freq);

    // Play Sample liefert den belegten
    // Kanal zurück
```

```
int voice = play_sample(sample,64, 128, 1000, 0);
timerCounter = 0;
syncTimer(&timerCounter);

while (!key[KEY_ESC]) {
    if (timerCounter) {
        while (timerCounter) {
            --timerCounter;
        }
        blit(bg,doubleBuffer,0,0,0,0,bg->w,bg->h);
        textprintf_centre(doubleBuffer, font, doubleBuffer->w/2,
        doubleBuffer->h - 20, makecol(255,255,255), text);
        textprintf(doubleBuffer, font, 20, 20,
        makecol(255,255,255), "%i", freq);
        needsRefresh = 1;

        // Ist der Kanal wieder frei?
        if (voice_check(voice) == NULL) {
            // Yep, Abspielen beendet
            freq += 10;
            createRectWave(sample, freq);
            voice = play_sample(sample,64, 128, 1000, 0);
        }
    }

    if (key[KEY_SPACE]) {
        if (!spaceDown) {
            spaceDown = 1;
            snd = rnd(3);
            // Abspielen stoppen
            stop_sample(sample);
            // und speichern
            export_sample("out.wav", sample);
        }
    } else {
        spaceDown = 0;
    }

    if (needsRefresh) {
        show();
        needsRefresh = 0;
    }
}

destroy_bitmap(bg);
destroy_sample(sample);
```

```
    done();
    return 0;
} END_OF_MAIN()
```

Spielen Sie mit diesem Programm etwas herum. Sie könnten zum Beispiel weitere Funktionen schreiben, die anstatt einer Rechteckschwingung eine Sinus- oder eine Sägezahnschwingung erzeugt.

Die BeatBox

Nun wird es Zeit, dass wir mit dem neu erworbenen Wissen bei einem interessanteren Projekt einsetzen. Eine »Beatbox« ist ein kleines Tool, mit dem man verschiedene Rhythmen erzeugen und speichern kann.

Es werden 16 verschiedene Soundsamples geladen. Jedes dieser Samples hat seine eigene Spur. Der gesamte Song besteht aus vier Takten. Je Takt haben Sie entweder 8, 16 oder 32 Schläge zur Verfügung. Änderungen, die im ersten Takt gemacht werden, werden in allen anderen Takten wiederholt.

Sie können auch durch einen einfachen Tastendruck einen zufälligen Rhythmus erstellen lassen. Sobald Sie einen interessanten Rhythmus kreiert haben, können Sie diesen in eine WAV-Datei exportieren.

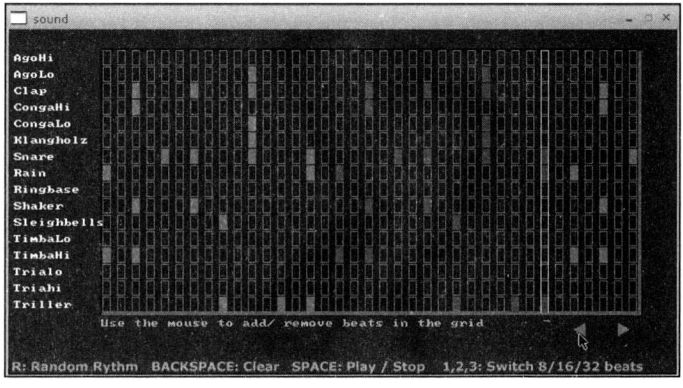


Abbildung 20.4: Die BeatBox

Die Oberfläche der BeatBox ist komplett innerhalb des Programms implementiert, die Allegro GUI wird nicht benutzt.

```
int main(int argc, char** argv) {
    int instr;
    int x,y,pos,a;
    int down =0;
    int playing = 0;
    int keyDown[256];
    int curBeat;
    int lastBeat;

    memset(keyDown  , 0, sizeof(int) * 256);
    cur_beats_per_beat = 16;

    init();
    if (!loadDrumSet("drums")) {
        allegro_message("danger!");
    }
    createRandomRythm();
    bg = load_bitmap("bg.tga", NULL);
    while (!key[KEY_ESC]) {
        /*clear(doublebuffer);*/
        blit(bg, doublebuffer,0,0,0,0,bg->w,bg->h);
        clear_to_color(gridbuffer,
                        bitmap_mask_color(gridbuffer));
        y = 0;
        for (instr=0; instr < COUNT_INSTR; instr++) {
            x = 0;
            for (pos=0; pos < BEATS_PER_LINE;
                            pos+=cur_beats_per_beat) {
                renderRythm(gridbuffer, instr, x, y,
                            gridbuffer->w, BEAT_H,
                            cur_beats_per_beat);
            }
            y+=BEAT_OFS;
        }
        if (mouse_b) {
            if (!down) {
                down = mouseClickHandler(
                                    mouse_x, mouse_y);
            }
        } else {
            delta = 1.0;
            down = 0;
        }
```

```
if (playing) {
    if (timerCounter < 0) {
        timerCounter = 0;
    }
    curBeat = timerCounter
              * (MAX_BEATS_PER_BEAT
              / cur_beats_per_beat)
              % BEATS_PER_LINE;
    if (curBeat != lastBeat) {
        for (a=0; a < COUNT_INSTR; a++) {
            if (getData(a, curBeat)) {
                play_sample(drums[a],
                            255,128,1000,0);
            }
        }
        lastBeat = curBeat;
    }
    rect(gridbuffer, curBeat * BEAT_W,
         0, (curBeat+1)*BEAT_W,
         gridbuffer->h, selectionColor);

}

for (a=0; a < COUNT_INSTR; a++) {
    textprintf(doublebuffer, font,
             START_INSTR_NAME_X,
             START_INSTR_NAME_Y
               +a* BEAT_OFS
               +(BEAT_OFS-ext_height(font))/2,
              makecol(255,255,255), "%s",
             drumNames[a]);
}
masked_blit(gridbuffer, doublebuffer, grid_ofs,
            0, START_GRID_X, START_GRID_Y,
            512, gridbuffer->h);
textprintf(doublebuffer, font, 89, 282,
           makecol(200, 200, 255),
           "Use the mouse to add/ "
           "remove beats in the grid");
scare_mouse();
blit(doublebuffer, screen, 0, 0, 0, 0,
     SCREEN_W, SCREEN_H);
unscare_mouse();

KEY_HANDLER(KEY_SPACE)
    if (!playing) {
```

```
                playing = 1;
                timerCounter = -1;
                curBeat  = 0;
                lastBeat = -1;
            } else {
                playing = 0;
            }
        KEY_HANDLER_END(KEY_SPACE)

        KEY_HANDLER(KEY_BACKSPACE)
            memset(data, 0, sizeof(int)
                    * COUNT_INSTR * BEATS_PER_LINE);
        KEY_HANDLER_END(KEY_BACKSPACE)

        KEY_HANDLER(KEY_R)
            createRandomRythm();
        KEY_HANDLER_END(KEY_R)

        KEY_HANDLER(KEY_1)
            cur_beats_per_beat = 8;
        KEY_HANDLER_END(KEY_1)
        KEY_HANDLER(KEY_2)
            cur_beats_per_beat = 16;
        KEY_HANDLER_END(KEY_2)
        KEY_HANDLER(KEY_3)
            cur_beats_per_beat = 32;
        KEY_HANDLER_END(KEY_3)

        KEY_HANDLER(KEY_S)
            saveSample("test.wav");
        KEY_HANDLER_END(KEY_S)

        KEY_HANDLER(KEY_F12)
            save_bitmap("screenshot.bmp", doublebuffer,
                        NULL);
        KEY_HANDLER_END(KEY_F12)
    }

    for (a=0; a < COUNT_INSTR; a++) {
        destroy_sample(drums[a]);
    }

    return 0;
}
END_OF_MAIN()
```

Nach dem Setzen des Grafikmodus (in `init()`) werden die Samples durch die Methode `loadDrumSet()` geladen. Diese sucht im angegebenen Verzeichnis nach Wave Files und einer `ini`-Datei, welche die Namen der Samples enthält. Beim Laden wird auch darüber Buch geführt, welches der geladenen Samples am längsten ist. Dies ist notwendig, um für ausreichend Platz im Sample zu sorgen. Ansonsten könnte ein kurzes Stück vom letzten Beat abgeschnitten werden.

Die Tastaturabfrage wird durch zwei Makros vereinfacht:

```
#define KEY_HANDLER(code)   if (key[code]) { \
            if (!keyDown[code]) {\
                keyDown[code] = 1;
#define KEY_HANDLER_END(code) }\
        } else {\
            keyDown[code] = 0;\
        }\
```

Dadurch können die Tastaturabfragen entprellt werden und auf jeden Tastendruck wird nur einmal reagiert.

Die BeatBox war am Anfang als reines C-Programm konzipiert. Aus diesem Grund sind viele Problemlösungen sehr C-lastig.

Bei einem Mausklick wird die Routine `mouseClickHandler()` aufgerufen, welche erst überprüft, ob der sichtbare Bereich gescrollt werden soll, und dann gegebenenfalls einen Beat setzt oder löscht.

```
int mouseClickHandler(int mx, int my) {
    int x     = (mx - START_GRID_X) + grid_ofs;
    int y     = (my - START_GRID_Y);
    int inc   = MAX_BEATS_PER_BEAT /
                                cur_beats_per_beat;
    int len   = (inc * BEAT_W);
    int pos   = x / len;
    int instr = y / BEAT_OFS;
    int ret   = 1;

    if (my >=SCROLL_LEFT_Y && my <=SCROLL_LEFT_Y+16) {
        if (mx >= SCROLL_LEFT_X
               && mx < SCROLL_LEFT_X+16) {
            grid_ofs = (int) (grid_ofs - delta);
            delta += 0.2;
            ret = 0;
        } else if (mx >= SCROLL_RIGHT_X && mx < SCROLL_RIGHT_X+16) {
```

```
            grid_ofs = (int) (grid_ofs + delta);
            delta += 0.2;
            ret = 0;
        }
        grid_ofs = MID(0, grid_ofs, gridbuffer->w-514);
    }

    if (mx >= START_GRID_X && my >= START_GRID_Y) {
        toggleData(instr, pos*inc);
    }
    return ret;
}
```

Die `toggleData()`-Methode überprüft ob im ersten Takt (Beat < 32) geklickt wurde. Ist dies der Fall, wird der Beat in allen Takten gesetzt, ansonsten nur in dem Takt, in dem geklickt wurde.

```
void toggleData(int instr, int pos) {
    int beat = !data[instr * BEATS_PER_LINE + pos];
    int a;
    if (pos < 32) {
        for (a=0; a < MAX_BEATS; a++) {
            data[instr * BEATS_PER_LINE
                    + (pos + MAX_BEATS_PER_BEAT *a)]
                = beat;
        }
    } else {
        data[instr * BEATS_PER_LINE + pos] = beat;
    }
  }
```

Wie Sie sehen eignet sich Allegro nicht nur für Spiele – es ist auch sehr einfach, Multimedia-Tools oder Hilfsprogramme für die Entwicklung von Spielen zu schreiben.

Den kompletten Quellcode finden Sie auf der beliegenden CD, im Unterordner *Beatbox* des Quellcode-Verzeichnisses für dieses Kapitel.

21 Titelbilder und Grafiken

Auch kleine Spiele benötigen heutzutage bereits eine Vielzahl von Grafiken. In diesem Kapitel gehen wir auf die in Spielen verwendeten Perspektiven ein und zeigen häufig genutzte Layouts für Titelbilder und Werbegrafiken.

Um gute Grafiken zu erstellen, braucht man neben dem Talent auch die Zeit seine Fähigkeiten in diesem Bereich zu verbessern. Zwar haben viele Entwickler eine gute Portion Talent, aber nur die wenigsten haben die Zeit ihre Fähigkeiten im künstlerischen Bereich zu verbessern.

Aus diesem Grund bestehen die meisten (Hobby-)Entwicklerteams aus zumindest einem Grafiker/Designer und einem Entwickler. Wenn man sich in seinem Bekanntenkreis umsieht, dann findet man normalerweise recht schnell jemanden, der »gut zeichnen« kann oder in seiner Freizeit gerne Computergrafiken erstellt. Und die Aussicht auf ein Spiel mit ihren Grafiken ist für viele Künstler auch ein hoher Anreiz.

Mit etwas Glück ist der von Ihnen Ausgewählte sogar jemand, der gerne Computerspiele spielt. In diesem Fall können Sie ihm über Vergleiche erklären, was Sie genau wollen. »Die Perspektive so wie in [Spiel1] aber von der Stimmung her viel dunkler, eher so wie in [Spiel2], aber mit besserer Animation«.

Mit ein paar Beispiel-Screenshots kann man dann meistens verständlich machen, was man braucht. Meistens ...

Der Punkt ist: Auch ein Grafiker braucht so viele Informationen, wie er bekommen kann. Je genauer Ihre Angaben sind, um so eher kann er sich ein Bild machen. Und nur wenn sein inneres Bild mit dem Ihren möglichst übereinstimmt, werden am Ende beide zufrieden sein.

In diesem Kapitel bekommen Sie einen kurzen Einblick in die technischen Details der Grafik – aber diesmal mehr aus künstlerischer Sicht.

Perspektive

Die Perspektive, das heißt der Blickwinkel einer Grafik, beeinflusst das Aussehen des Spieles immens. Spiele nutzen häufig nicht die normale, realistische Perspektive, sondern Abwandlungen, die es dem Programmierer einfacher machen ,die Grafiken einzusetzen.

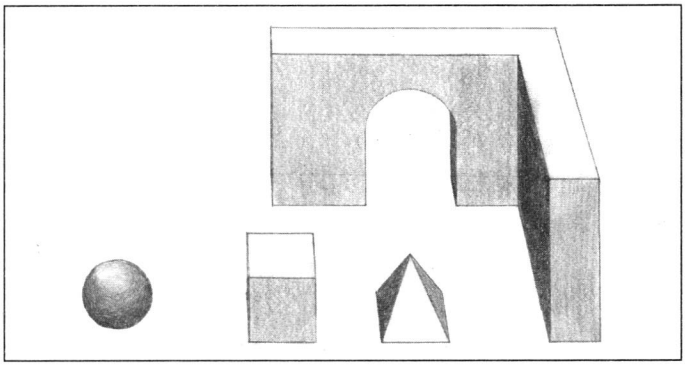


Abbildung 21.1: Die Beispielszene

Die Prinzipien der einzelnen Blickweisen werden an folgenden Grundkörpern erklärt:

- ✔ Kugel
- ✔ Würfel
- ✔ Pyramide
- ✔ Mauer mit Durchgang

Zwar ist eine Mauer kein Grundkörper im geometrischen Sinn, aber es handelt sich durchaus um einen Grundkörper aus spieltechnischer Sicht, da Sie Mauern und Durchgänge häufiger brauchen werden.

Draufsicht

Bei der *Draufsicht* (englisch: *Top-Down View*) handelt es sich um eine der einfachsten Ansichten. Von allen Gegenständen wird nur die von oben sichtbare Fläche dargestellt. Keine der Seitenwände ist sichtbar (siehe Abbildung 21.2).

Das Hauptproblem bei dieser Perspektive ist die Tatsache, das der Durchgang nicht mehr sichtbar ist. In solchen Fällen können Sie entweder den Bereich über dem Durchgang schmaler machen oder durch eine Veränderung des Bodens den Durchgang anzeigen. Häufig wird der Boden aufgehellt, um das durch den Durchgang schimmernde Licht zu simulieren. In Abbildung 21.3 sehen Sie einen Durchgang, bei dem beide Methoden angewandt wurden.

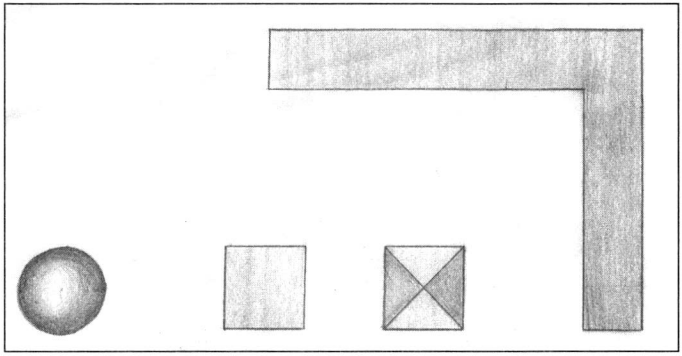


Abbildung 21.2: Draufsicht

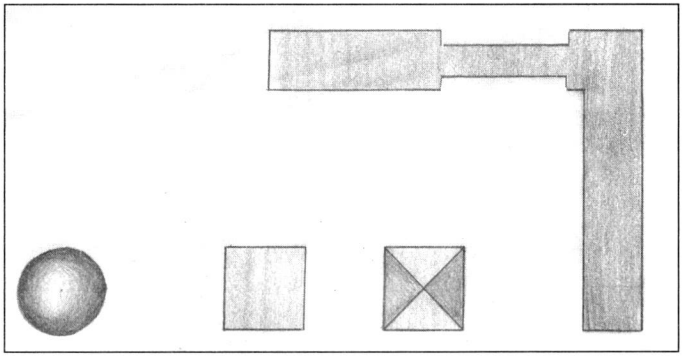


Abbildung 21.3: Abgewandelte Top-Down View

Neben dem Lichtschein können Sie auch Teppiche oder einen andersfarbigen Teppich benutzen.

Seitenansicht

Die *Seitenansicht* (englisch: *Side View*) ähnelt der Draufsicht. Auch hier wird nur eine Seite der Objekte gezeigt und zwei gleich große Objekte haben unabhängig von ihrer Nähe zum Betrachter auch immer die gleiche Größe.

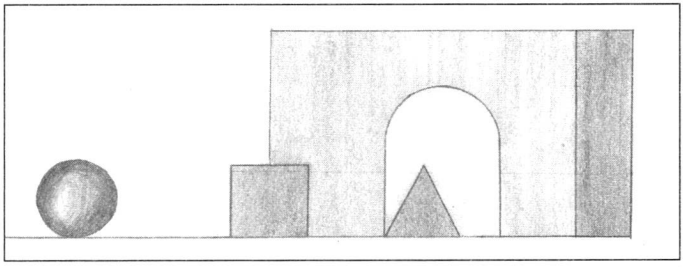


Abbildung 21.4: Seitenansicht

In dieser Ansicht gibt es nur selten Probleme. Sollen Durchgänge angezeigt werden, so wird dies in der Regel durch ein anderes Material (das heißt: Farbe und Muster des Durchgangs) angezeigt, oder man macht in der Mitte des Durchgangs einen »Schnitt« durch die Wand.

Die Seitenansicht wird in erster Linie bei Hüpf- und Plattformspielen verwendet. Auch einige Ballerspiele lassen Ihre Hauptfigur in dieser Ansicht durch die Gegend eilen.

Klapp-Perspektive

Die Klapp-Perspektive ist eine Kombination von Drauf- und Seitenansicht. Da es eine solche Ansicht in der realen Welt nicht gibt, bezeichnet man die Klapp-Perspektive auch als Pseudoperspektive.

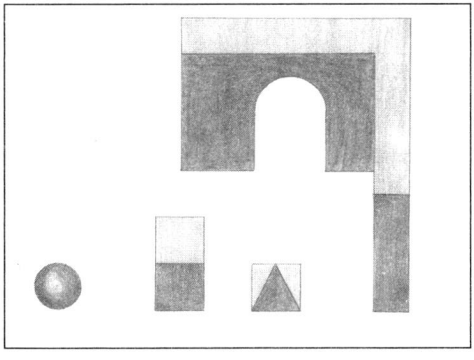


Abbildung 21.5: Die Klapp-Perspektive

Man sieht sowohl die Vorderseite, als auch die Oberseite eines Objektes. Dadurch sind die Eingänge an Häusern gut sichtbar, und auch Höhenunterschiede lassen sich gut darstellen.

Diese Perspektive wird sehr gerne in japanischen Rollenspielen verwendet.

Kavalierperspektive

Die Kavalierperspektive gehört auch zu den Pseudoperspektiven. Man sieht die Vorderansicht der Objekte und die Seitenansicht. Alle nach hinten gehenden Linen werden in einem Winkel von 45 Grad und um die Hälfte verkürzt gezeichnet.

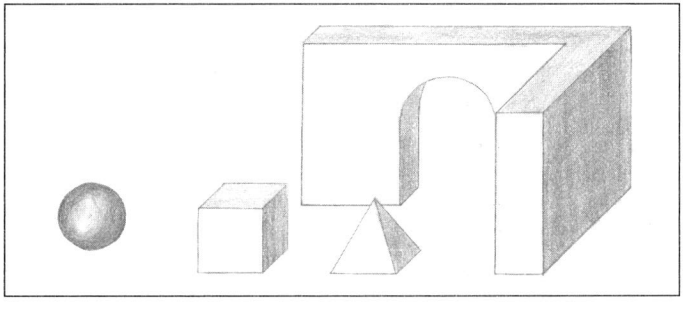


Abbildung 21.6: Kavalierperspektive

Diese Perspektive wird kaum in Spielen eingesetzt, aber in den Spielen, welche die Kavalierperspektive nutzen, macht sie eine sehr gute Figur.

Isometrische Perspektive

Die isometrische Perspektive stellt Objekte um 45 Grad gedreht dar. Eine der Seitenkanten zeigt in Richtung des Betrachters, die von dieser Kante nach hinten gehenden Linien werden in einem Winkel von 30 Grad zur Kante dargestellt.

Die Isometrie-Perspektive ist in Spielen sehr beliebt, da sie zum einen sehr realistisch wirkt, zum anderen aber die Vorteile der einfacheren Perspektiven größtenteils erhalten bleiben.

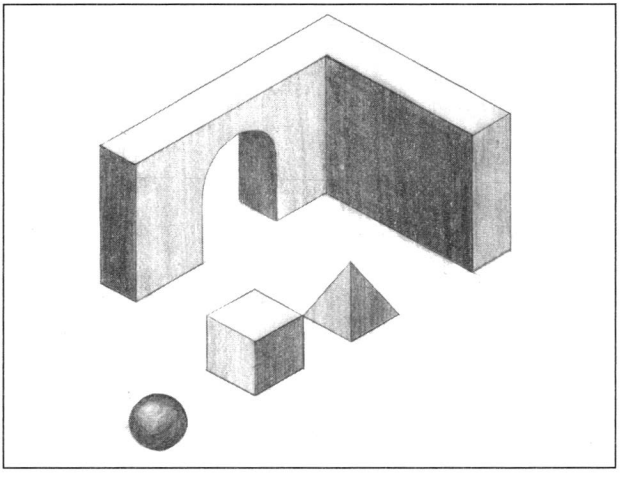


Abbildung 21.7: Isometrische Ansicht

Zentralperspektive

Bisher waren die vorgestellten Ansichten alle orthografisch. Bei einer orthografischen Perspektive hat die Entfernung des Objekts zum Betrachter keinen Einfluss auf die Größe, in der das Objekt dargestellt wird.

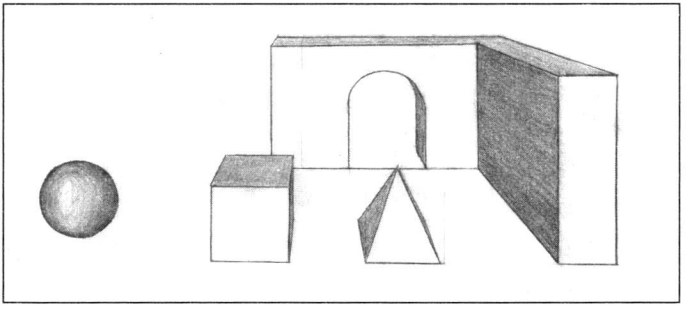


Abbildung 21.8: Zentralperspektive (1 Fluchtpunkt)

Bei der Zentralperspektive ist es nun so, dass Objekte, die weiter entfernt sind, auch kleiner dargestellt werden.

Die Konstruktion einer Zentralperspektive basiert auf der Idee eines *Fluchtpunkts*. Alle in den Raum gehenden Linien werden in Richtung dieses Fluchtpunktes gezeichnet. Zieht man eine waagerechte Linie durch diesen Punkt, so erhält man die Horizontlinie.

Vogelperspektive

Auch bei der Vogelperspektive handelt es sich um eine Perspektive mit einem einzelnen Fluchtpunkt. Aber diesmal werden alle nach unten gehenden Linien in Richtung des Fluchtpunktes gezeichnet.

Vergleichen Sie die Vogelperspektive mit der Draufsicht. Obwohl der Blickwinkel identisch ist, sind die resultierenden Grafiken sehr unterschiedlich.

Zwei-Fluchtpunktperspektive

Man könnte sagen, dass es sich bei der Perspektive mit zwei Fluchtpunkten um die realistischere Variante der isometrischen Perspektive handelt. Auch hier werden senkrechte Linien unverändert übernommen, allerdings werden die in den Raum gehenden Linien nicht in einem festen Winkel gezeichnet, sondern in Richtung einer der Fluchtpunkte.

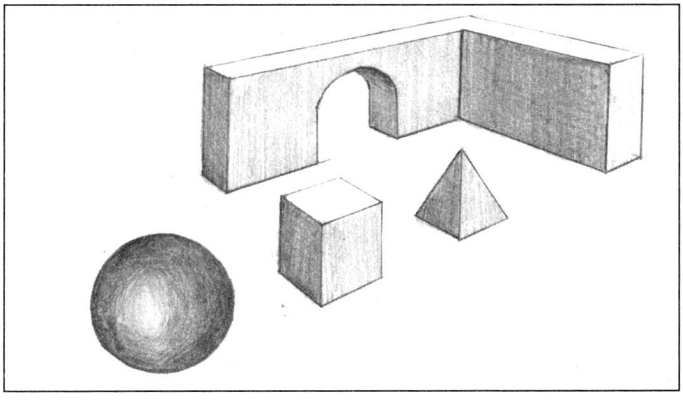


Abbildung 21.9: Zwei Fluchtpunkte

Entwurf von Titelbildern

Das Titelbild vermittelt den ersten Eindruck an den Spieler. Ein gutes Bild macht ihn neugierig und gibt ihm hoffentlich einen guten Eindruck von den auf ihn wartenden Aufgaben.

Titelbilder in Spielen sind Kinoplakaten sehr ähnlich. Und wenn sich einmal einige Kinoplakate ansehen, dann werden sie feststellen, dass sich bestimmte Anordnungen ständig wiederholen. Ein Katalog mit Kinopostern kann eine wertvolle Anregung sein, da Sie hier unzählige Poster ansehen und vergleichen können. Und natürlich ist der Katalog deutlich günstiger als sich all diese Poster zu kaufen.

Neben Kinopostern können auch Cover von Zeichentrickfilmen und (natürlich) Computerspielen eine Anregung bieten.

Drei Helden sollt Ihr sein

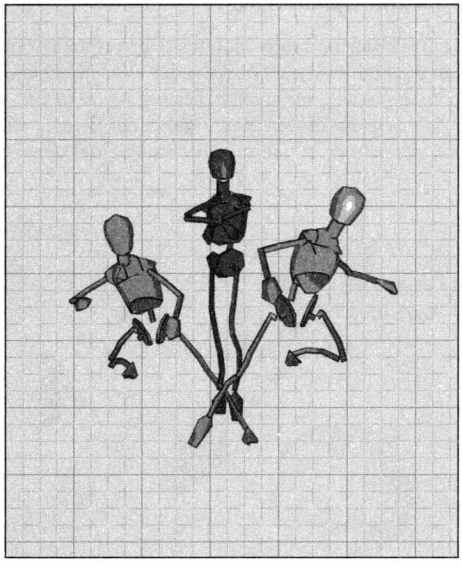


Abbildung 21.10: Titel-Layout: Action-Adventure (Drei Helden)

Bei diesem Layout liegt die Betonung auf den Charakteren. Der stärkste der drei ist der auf der rechten Seite. Dann folgt der Charakter auf der linken. Die Figur im Hintergrund ist entweder der Auftraggeber (dann passt die dargestellte Pose), das »Hirn« der Gruppe (in diesem Fall sollte er nachdenklicher wirken, zum Beispiel durch einen an das Kinn gelegten Finger) oder der Magier. Der Zauberwirker sollte immer in Aktion gezeigt werden, zum Beispiel mit nach oben gereckten Armen, um welche sich magische Energie sammelt, oder beim Blick in eine Kristallkugel.

Kämpfer und Magier

Während der Kämpfer gebückt und bereit in Aktion zu treten dasteht, ist der Magier ein Pol der Ruhe. Er kann entweder schweben, seinen Zauberstab nach oben recken oder einfach nur ruhig dastehen. Im letzten Fall sollte aber irgend etwas auf die magischen Fähigkeiten hindeuten. So könnten seine Augen blitzen oder er von einer feurigen Aura umgeben sein.

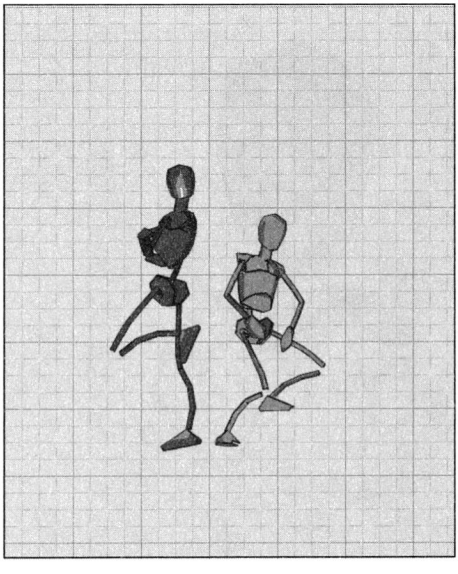


Abbildung 21.11: Titel-Layout: Action-Adventure (Zwei Helden)

Wenn Ihr Spiel nicht in einer Fantasy-Umgebung spielt, sondern in einer Cyberpunk-Welt, dann könnte die Rolle des Magiers von einem Hacker übernommen werden. In diesem Fall ist es genauso wichtig, dass Sie die besonderen Fähigkeiten darstellen. So könnte er teilweise als Drahtgittermodell gezeigt werden, oder mit einer Tastatur im Arm – die durch ein Kabel mit seinem Kopf verbunden ist.

Der Gegner im Hintergrund

Bei diesem Layout liegt die Betonung auf dem bedrohlich im Hintergrund schwebenden Bösewicht.

Die Hauptfiguren stehen im Vordergrund und scheinen bisher noch nichts von der bösen Präsenz, die sie beobachtet, zu ahnen. Variationen dieses Themas sind Hexenmeister, welche die Helden in einer Kristallkugel beobachten oder mit ihren klauenähnlichen Händen nach ihnen greifen.

Die Farbgebung sollte den bedrohlichen Eindruck noch verstärken.

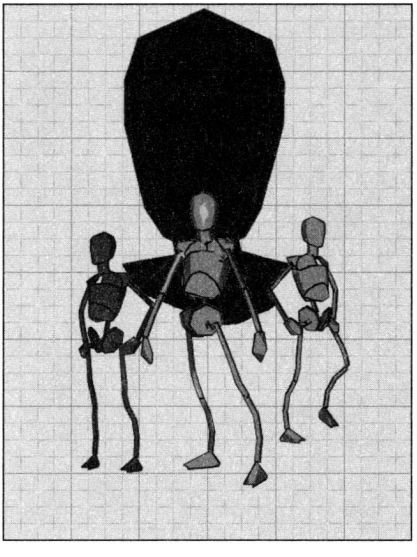


Abbildung 21.12: Titel-Layout: Drei Helden vor Gegner im Hintergrund

Die Heldensaga

Wenn Sie ein Spiel über einen Helden machen (wie Herakles, Odysseus oder sogar Götter wie Thor), dann sollte dieser deutlich im Vordergrund stehen und die anderen Personen in einer Art Dreieck hinter ihm gruppiert sein.

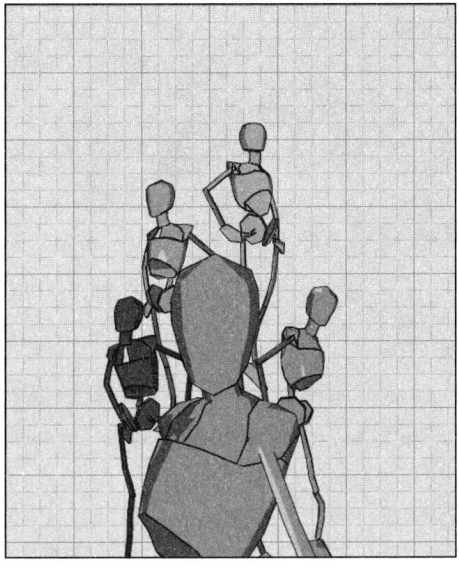


Abbildung 21.13: Titel-Layout: Heldensaga (ein Held + vier NPCs)

Hier können sowohl gute als auch böse Nebencharaktere gezeigt werden. Aber stellen Sie durch geeignete Mimik klar, wer zu den Guten und wer zu den Bösen gehört.

Der einzelne Held in Bewegung

Dieses Layout wird sehr gerne für Hüpfspiele verwendet. Im Hintergrund werden entweder kräftige Farben (zum Beispiel eine Explosion oder farbige Bewegungslinien) oder ein stilisierter Levelaufbau verwendet.

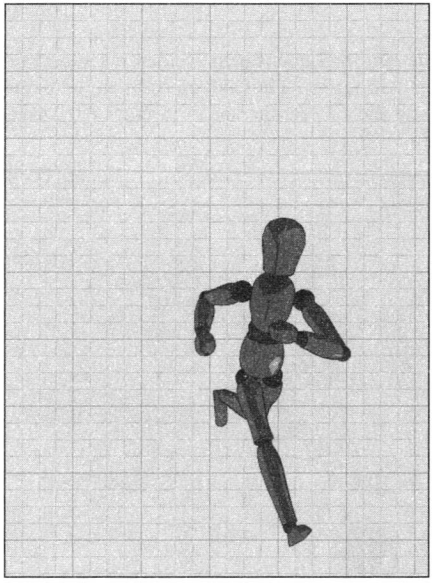


Abbildung 21.14: Titel-Layout: Action-/Platform Game

Das Action-Spiel

Liegt der Schwerpunkt klar auf den Actionelementen, dann platzieren Sie Ihre Hauptfigur in einer starken Pose an einem der äußeren Ränder. Im Hintergrund stehen dann einige Actionelemente (Kampfszenen für ein Prügelspiel, Raumschiffe bei einem Weltraumballerspiel und so weiter).

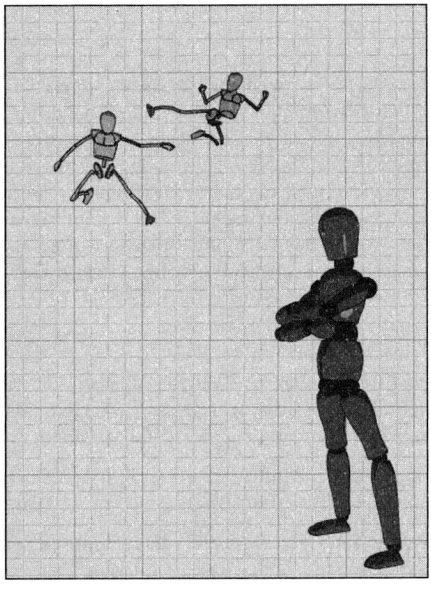


Abbildung 21.15: Titel-Layout: Action Game

Weitere Ideen

Die hier gezeigten Layouts sind natürlich nur Anregungen. Es gibt keine starren Regeln für Titelbilder. Wenn Sie eine gute Idee haben, zögern Sie nicht diese zu verwenden.

Auch können Sie die Elemente natürlich beliebig kombinieren. So können Sie im Hintergrund des »Heldensaga«-Layouts auch den Gegner positionieren. Auch ist die Anzahl der Figuren auf den Bildern natürlich nur eine Richtlinie.

Nehmen Sie die hier gezeigten Bilder nur als Anregungen, nicht als in Stein gehauene Regeln.

Farben

Die in Ihrem Spiel verwendeten Farben können den Eindruck des Spieles erheblich beeinflussen. Farben können Gefühle wecken, sie können Erinnerungen wach rufen und sie haben eine bestimmte Bedeutung in unserer Kultur. Das Problem ist nur, dass Farben von verschiedenen Personen zum Teil unterschiedlich wahrgenommen werden und in verschiedenen Kulturen auch unterschiedliche Bedeutungen haben können. Gerade wenn Sie vorhaben Ihr Spiel über das Internet zu verteilen, sollten Sie sich über die Bedeutung der Farben ein paar Gedanken machen.

Farbtemperatur

Man sagt, Farben haben eine gewisse Wärme. Rot-, Gelb- und Orangetöne werden als warm und aufregend wahrgenommen. Blau-, Grün und Violetttöne hingegen sind eher kühl und gedämpft.

Mit warmen Farben können Sie ein Gefühl der Nähe und Freundlichkeit erzeugen oder ein Bild dramatisch erscheinen lassen.

Kühle Farben hingegen vermitteln den Eindruck von Entfernung, Leere und Weite.

Sie können dies dazu benutzen, um Ihren Bildhintergründen etwas mehr Tiefe zu verleihen. Nehmen wir an, Ihr Held kämpft sich durch ein Höhlensystem. Die ihn umgebenden Steine sind in neutralen Grautönen gehalten. Der Eindruck, der jetzt dem Spieler vermittelt wird ‚ist: Langeweile. Eine graue Umgebung wirkt dröge und uninteressant.

Wenn Sie jetzt die Tönung der Steine leicht in Richtung Orange verschieben, dann wirkt die Höhle auf einmal warm. Wenn Sie jetzt rote Lichtreflektionen auf einigen der Steine aufbringen, wird der Eindruck einer Lavahöhle oder eines Vulkans erzeugt.

Verschieben Sie die Tönung jedoch in Richtung Blau, dann wirkt die Höhle auf einmal kälter. Je nachdem wie stark Sie den Blauton wählen kann sogar der Eindruck einer Eishöhle erzeugt werden.

Wenn Sie den Steinen einen Braunton geben, dann ist das Ergebnis weicher, da wir Braun mit Erde verbinden – und Erde ist ein weicher, formbarer Gegenstand. Sie können hier wieder mit Rot- und Orangetönen bzw. mit Blau- und Violettönen die Wärme der Umgebung beeinflussen.

Assoziationen mit Farben

Farben wecken in uns Gefühle und Eindrücke. In der folgenden Tabelle finden Sie ein paar der geläufigeren Begriffe, die mit Farben verknüpft werden.

Farbe	Verknüpfte Begriffe
Rot	Gefahr, Hingabe, Aggression, Hitze, Liebe, Dringlichkeit, Blut
Grün	Sicherheit, Natur, Gesundheit, Glück, Neid
Blau	Kälte, Depression, Kühlheit, Gelassenheit, Friede
Gelb	Vorsicht, Wärme, Helligkeit
Magenta	Vorstellungsvermögen
Pink	Freundlich, niedlich, Romanzen
Orange	Warnung, Mut, Freude
Gold	Reichtum, Weisheit
Braun	Stabilität, Erde
Türkis	Erfrischend, Kühl
Purpur	Reichtum, Adel, Geheimnisse, Intelligenz
Grau	Neutralität, Schwermut
Schwarz	Tod, Böses, Stärke, Furcht
Beige	Neutral, Zerbrechlich
Weiß	Reinheit, Licht, Leere

Tabelle 21.1: Mit Farben verknüpfte Bedeutungen (Westeuropa)

Je nach kulturellem Hintergrund sind die mit den Farben verbundenen Bedeutungen zum Teil sehr unterschiedlich. Die Liste in Tabelle 21.1 bezieht sich auf Westeuropa.

Internationale Bedeutung von Farben

Während die Farbe Rot bei uns mit »Gefahr und Aggression« verbunden wird, so verbinden die Chinesen »Freude und Festlichkeiten« damit. »Grün vor Neid« ist bei uns eine feste Redensart, im Mittleren Osten ist Grün die Farbe der Fruchtbarkeit und Stärke. Weiß ist bei uns die Farbe der Reinheit und Unschuld, in Japan die Farbe des Todes und der Trauer.

In Tabelle 21.2 finden Sie eine Übersicht über die Bedeutung von Farben in einigen Ländern und Regionen.

Farbe	Westeuropa, USA	Japan	China	Mittlerer Osten
Rot	Gefahr, Wut, Stopp	Gefahr, Wut	Freude, Feierlichkeiten	Gefahr, Böses
Gelb / Gold	Vorsicht, Feigheit, Wärme	Freude, Eleganz, Edel	Ehre, Adel	Freude, Wohlstand
Grün	Neid, Sicherheit, Natur	Zukunft, Jugend, Energie	Hoffnung, Wohlstand	Fruchtbarkeit, Stärke
Weiß	Reinheit, Unschuld	Tod, Trauer	Trauer, Demut	Reinheit, Trauer
Blau	Ruhe, Männlichkeit	Gegner, Kälte	Stärke und Kraft	Schutz
Schwarz	Tod, Böses	Böses	Böses	Geheimnis, Böses

Tabelle 21.2: Bedeutung von Farben international

Natürlich stellen diese Tabellen nur Anregungen dar. Über die eigentliche Farbgebung entscheiden natürlich nur Sie selbst.

22 Das Veröffentlichen von Spielen

Wie kommt es, dass einige Spiele kaum beachtet werden, andere sich aber mit enormer Geschwindigkeit verbreiten? Was macht den Unterschied aus zwischen einem Spiel, das »weg geht wie warme Semmeln« und einem, das kaum bemerkt wird? In den meisten Fällen liegt es schlicht an der Werbung, die für ein Spiel gemacht wird. Auch eine Rolle spielt, wie einfach oder schwer es für den Spieler ist, an das Programm heranzukommen.

Gerade bei kleinen Spielen, die man mal eben in einer Pause zwischendurch zockt, ist die Qualität nicht mehr so entscheidend (solange das Programm keinen Absturz verursacht.

Wenn Sie ein Programm fertiggestellt haben, dann sollten Sie es nicht überstürzt auf Ihre Webseite hochladen, sondern genau planen, wie Sie Ihr Spiel bekanntgeben möchten. In diesem Kapitel zeige ich Ihnen, wie das funktioniert.

Checkliste

Doch bevor es soweit ist, gehen Sie nochmals folgende Liste durch:

- ✔ Das Spiel wurde auf mehreren (mindestens drei) unterschiedlichen Rechnern getestet. Nach Möglichkeit sollten auf diesen Rechnern auch unterschiedliche Betriebsysteme laufen (Windows 98, Windows NT, Windows XP, SuSE Linux, Redhat Linux, Debian, BSD etc.).
- ✔ Sie kennen die Mindest- und Idealanforderungen Ihres Spieles (CPU, Festplattenspeicher, Arbeitsspeicher, Videospeicher).
- ✔ Sie haben eine Webseite vorbereitet, auf der Informationen zum Spiel und Bildschirmfotos abrufbar sind.
- ✔ Sie haben ein Installationsprogramm, das dem Nutzer die Arbeit abnimmt.
- ✔ Sie kennen die mögliche Zielgruppe Ihres Spieles.
- ✔ Sie haben eine *ReadMe.txt*-Datei geschrieben.

Dass Sie Ihr Spiel auf möglichst vielen unterschiedlichen Rechnern testen, ist wirklich sehr wichtig. Fragen Sie in einem Programmiererforum, ob jemand Interesse hat Ihr Spiel zu testen. Fragen Sie Freunde und Bekannte.

Wenn Sie die Chance dazu haben, dann lassen Sie einen Ihrer Bekannten einmal das Spiel spielen. Geben Sie ihm keine Hinweise, sondern lassen Sie ihn das Spiel entdecken. Wo hat er Schwierigkeiten? Ist ihm sofort klar, was er machen muss?

Wenn Sie ihm durch das Hauptmenü helfen müssen und ihm nicht klar ist, was er eigentlich tun soll, dann ist die Chance groß, dass es anderen Leuten genauso geht.

Und wie lange würden Sie ein Spiel spielen, von dem Sie nicht viel wissen, und bei dem Sie keine Ahnung haben, worum es eigentlich geht? Wenn Ihre Antwort auf diese Frage »nicht lange« ist, dann geht es Ihnen ebenso, wie der Mehrzahl der Computernutzer weltweit. Die einzige Chance, dass jemand ihr Spiel nach noch kürzerer Zeit wieder deinstalliert ist, wenn es

- ✔ gar nicht erst startet,
- ✔ es mit einem Fehler abstürzt.

Und wenn Sie der Meinung sind, dass ein Spieler erst einmal die Anleitung (oder die *ReadMe.txt*-Datei) durchliest, bevor er das Spiel startet, dann haben Sie noch nie Hunderte von E-Mails bekommen, die genau nach den Dingen fragen, die in der *ReadMe* erklärt werden.

Heutzutage kann man sich beliebig viele Spiele und Demos aus dem Netz herunterladen. Der Spieler kann wählerisch sein, und glauben Sie mir: Er ist es. Aus diesem Grund sollte es dem Spieler so einfach und mühelos wie möglich gemacht werden.

Wenn Ihr Spiel unter einer der Windowsplattformen läuft, dann sollten Sie sowohl eine selbstinstallierende EXE-Datei als auch ein ZIP-File bereitstellen. Einige User bevorzugen das einfache Durchklicken durch ein Installationsprogramm, andere schrecken davor zurück, dass »irgendein Programm Werte in meine Registry schreibt«. Später in diesem Kapitel gehe ich kurz auf Inno Setup ein. Inno Setup ist ein Freeware Tool, das es Ihnen erlaubt, mit sehr wenig Aufwand professionell wirkende Installationsprogramme zu schreiben.

Sie müssen die Zielgruppe Ihres Spieles kennen, um entscheiden zu können, auf welchen Webseiten Sie dafür Werbung machen bzw. wo Sie es

bekanntgeben. Ein witziges Puzzlespiel wird auf einer Webseite für Hardcore-Gamer untergehen, kann aber auf einer Portalseite zum Downloadmagnet werden.

Wenn Sie ein Quizspiel geschrieben haben, dann nehmen Sie sich zum Beispiel die Zeit nach Internetseiten zu suchen, die sich mit Quizshows und Knobeleien befassen. Schreiben Sie mal den Webmastern dieser Seiten, und geben Sie Ihnen freundliches Feedback. Knüpfen Sie Kontakte. Sobald Ihr Spiel dann zum Download bereitsteht, schreiben Sie diesen Webmastern nochmals eine Mail und fragen Sie, ob Sie eventuell eine News Meldung darüber veröffentlichen könnten. Und schon haben Sie Ihr Spiel im Blickfeld der potentiellen Spieler platziert.

Wenn Sie ein Prügelspiel geschrieben haben, dann gehen Sie vom Prinzip her genauso vor, aber dieses Mal sollten Sie schon einige Zeit, bevor Sie das Spiel veröffentlichen, Nachrichten in den Schwarzen Brettern zum Thema Prügelspiele hinterlassen. So können Sie zum einen herausfinden, welche Features besonders wichtig sind – aber viel wichtiger ist, dass die regelmäßigen Poster in diesem Board sich an Ihren Namen erinnern. Zögern Sie nicht, ein paar dieser Leute als Beta-Tester für Ihr Spiel zu rekrutieren. Und wenn das Spiel fertig ist, dann schreiben Sie eine Nachricht in das Board. Da sich die Beta-Tester als Teil des Entwicklungsprozesses fühlen, werden Sie gerne mithelfen, das Spiel bekannt zu machen.

Das Installationsprogramm

Mit einem guten Installationsprogramm erhält Ihr Spiel gleich eine professionellere Note. Das Freeware-Programm »Inno Setup« ist in Verbindung mit dem Hilfsprogramm »IS Tool« der einfachste Weg zu einem guten Setup-Programm. Auf der Inno Setup Webseite (*http://www.jrsoftware.org/isinfo.php*) finden Sie neben weiteren Informationen auch den Link auf die Webseite von IS Tool. Installieren Sie beide Programme und nehmen Sie sich etwas Zeit die Anleitungen zu lesen. Mit Hilfe dieser beiden Tools beschränkt sich die Arbeit zum Erstellen des Installers auf ein paar Klicks und die Auswahl der nötigen Dateien.

Starten Sie Inno Setup und wählen Sie *File/ New* (siehe Abbildung 22.1).

Geben Sie auf der nächsten Seite den Namen, die Version und die Webseite für das Programm ein (siehe Abbildung 22.2).

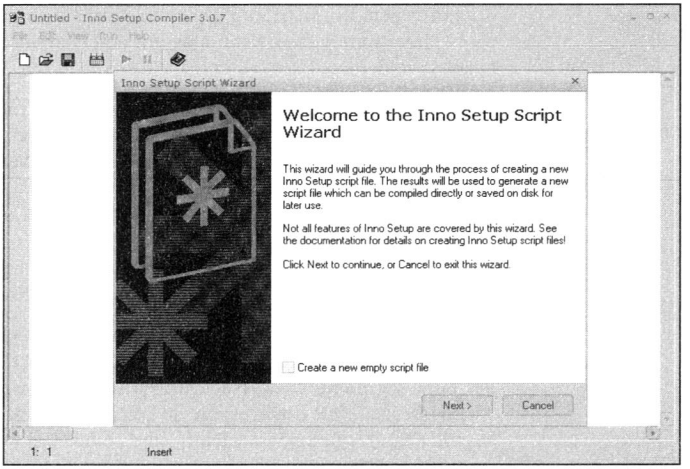


Abbildung 22.1: Der Inno-Setup-Wizard

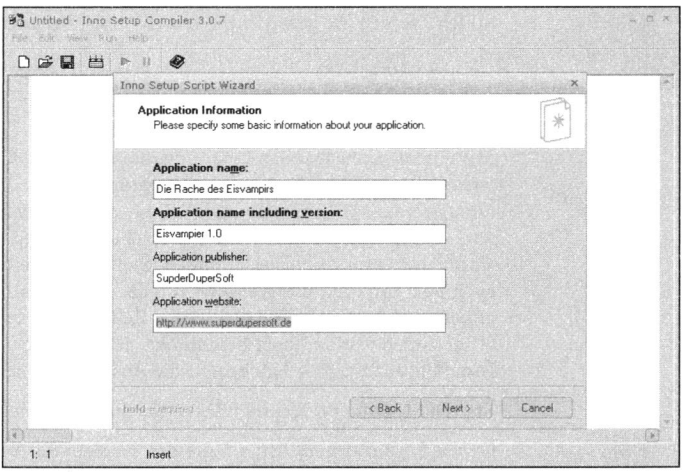


Abbildung 22.2: Setup-Wizard-Informationen über das Programm

Auf der nächsten Seite können Sie normalerweise alle Informationen übernehmen.

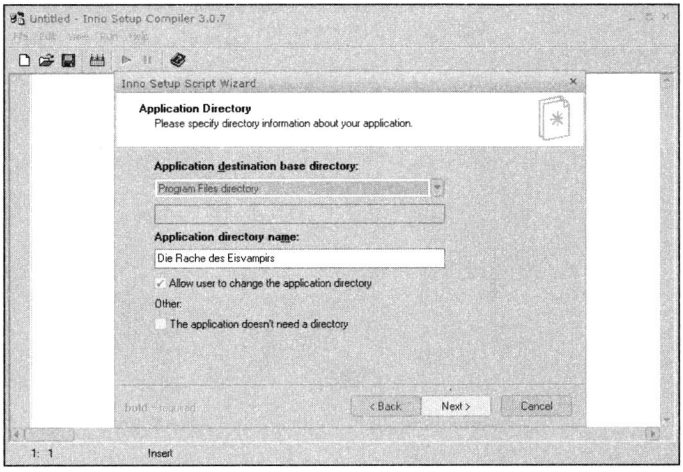


Abbildung 22.3: Setup Wizard – Directoryeinstellungen

Auf der nächsten Seite sollten Sie alle Dateien auswählen, die für Ihr Programm benötigt werden (siehe Abbildung 22.4).

Auf der nächsten Seite, Application Icons, sollten Sie zusätzlich zu den ausgewählten Optionen auch noch *Create Internet shortcut in the Start Menu Folder* und *Create Uninstall icon in Start Menu Folder* auswählen, damit sowohl Ihre Webseite als auch das Uninstall-Programm über das Startmenü erreichbar sind (siehe Abbildung 22.5).

Auf der nächsten Seite können Sie drei Textfiles auswählen. Die Lizenzdatei (Licence File) wird vor der Installation angezeigt, und nur nach Bestätigung der Lizenz kann das Setupprogramm fortgesetzt werden. In der zweiten Zeile können Sie eine Informationsdatei anzeigen. Allerdings sollten Sie sich nicht darauf verlassen, dass diese Datei dann auch wirklich vom Spieler gelesen wird. Sie können auch eine Datei auswählen, die nach Beenden der Installation angezeigt wird. Bei dieser stehen die Chancen etwas besser, dass sie gelesen wird. Aber auch darauf sollten Sie sich nicht verlassen (siehe Abbildung 22.6).

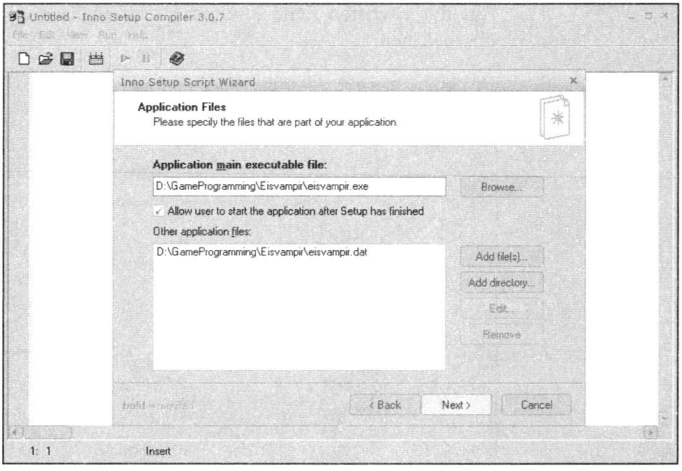


Abbildung 22.4: Setup-Wizard – Benötigte Dateien

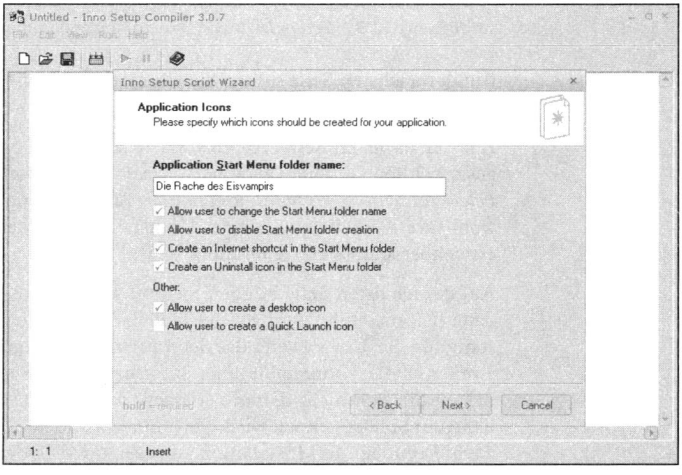


Abbildung 22.5: Setup-Wizard – Start-Menü-Einträge

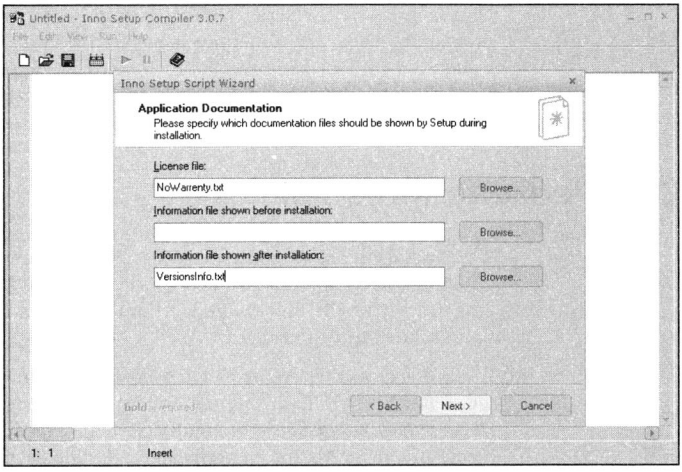


Abbildung 22.6: Setup-Wizard – Informationsdateien

Noch ein paar Klicks und Ihr Setupprogramm ist fertig!

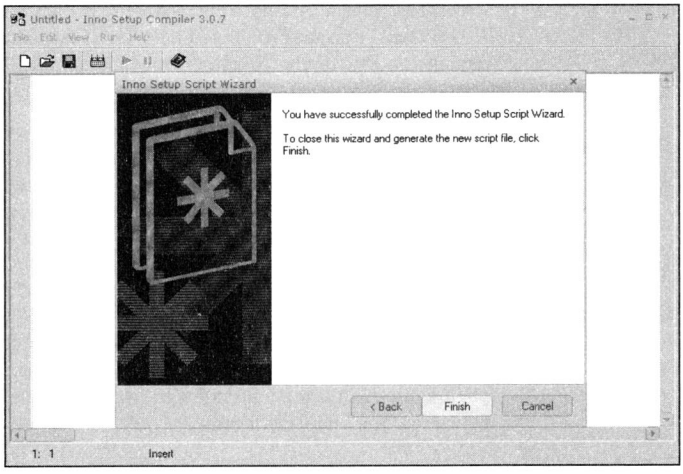


Abbildung 22.7: Setup-Wizard – Abschluss

Nun können Sie Ihr Script kompilieren und erhalten ein fertiges Setupprogramm. Falls Sie Änderungen am Script vornehmen wollen, dann sollten Sie IS Tool benutzen, da es Ihnen eine gute Übersicht über die Einstellungen liefert, und Sie bequem Änderungen und Ergänzungen vornehmen können.

Die Webseite Ihres Spieles

Wählen Sie ein klares, gut lesbares Design. Sie können sich an den Webseiten größerer Spielzeughersteller orientieren. Wichtig ist vor allem, dass der potentielle Spieler die von ihm gesuchten Informationen möglichst schnell finden kann.

In der Regel wird er erst nach einer kurzen Beschreibung Ausschau halten, dann nach Bildschirmfotos und dann entweder nach einem FAQ (***F**requently **A**sked **Q**uestions* = Häufig Gestellte Fragen) um mögliche Probleme direkt im Voraus zu vermeiden oder gleich nach dem Download Link.

Ich würde Ihnen aus diesem Grund vorschlagen, diesen Weg direkt zu unterstützen. So könnten Sie auf der Infoseite einen einzelnen Screenshot platzieren und daneben einen Link auf die Seite mit weiteren Screenshots setzen. Auf der Downloadseite sollten Sie einen Link auf das FAQ setzen – und im FAQ eine Frage wie »Wo bekomme ich die neueste Version von Programm- X« mit einem Link auf die Downloadseite beantworten.

Bei der Seite mit den Bildschirmfotos sollten Sie besonders darauf achten, dass Sie schnell geladen wird. Benutzen Sie auf jeden Fall kleine Voransichten der eigentlichen Bilder, anstatt das komplette Bild anzuzeigen. Ich würde Ihnen auch empfehlen, zu jedem Screenshot einen kurzen Spruch zu schreiben. Dadurch wird dem Betrachter schneller klar, was er eigentlich sieht – und Sie können hier auch versuchen die Neugier zu wecken. Wenn Sie ein Bildschirmfoto von einem Ork haben, der gerade mit Ihrem Helden kämpft, dann weckt eine Bildunterschrift wie »Was dieser riesige Ork wohl bewacht?« Erwartungen in dem Besucher. Er bekommt das Gefühl, dass das Spiel mehr ist, als er auf diesen Screenshots sehen kann. Und um herauszufinden, was der Ork bewacht, muss er natürlich das Spiel herunterladen.

Denken Sie immer daran: Sie wollen nicht das Spiel präsentieren. Sie wollen den Besucher Ihrer Webseite dazu bringen, Ihr Spiel herunterzuladen und zu testen.

Bekanntmachung des Spieles

Das Spiel ist getestet, die Webseite ist bereit, und Sie wissen, welche Zielgruppe Sie ansprechen wollen. Was nun?

Bereiten Sie einen kurzen Text vor, eine Art Pressemitteilung, wenn Sie so wollen. Angenommen, Sie wollen das Spiel »Rachedurst des Eisvampirs«, ein Action-Rollenspiel, bekannt machen. Ein Text könnte das Spiel wie folgt beschreiben:

»Ein Action-Rollenspiel der Oberklasse: SuperDuperGames haben ihr Spiel *Rachedurst des Eisvampirs* veröffentlicht. Schlüpfen Sie in die Gestalt von Lars Lavason und helfen Sie ihm, seine Schwester aus den Klauen des grausigen Eisvampirs zu befreien. 20 Level, 100 verschiedene Monster und 50 Zaubersprüche machen dieses Spiel zu einem Muss für jeden ernsthaften Spieler. Die kostenlose Probe-/Vollversion finden Sie auf: *http://ich.habe.eine.webseite.de*.«

Schicken Sie diesen Text mit der Bitte um Veröffentlichung an die Spieleseiten, die viele tägliche Besucher haben. Denken Sie auch daran, dass viele Fernsehsender und Webmail-Provider eigene Portalseiten für Spiele haben. Schreiben Sie auch diese an.

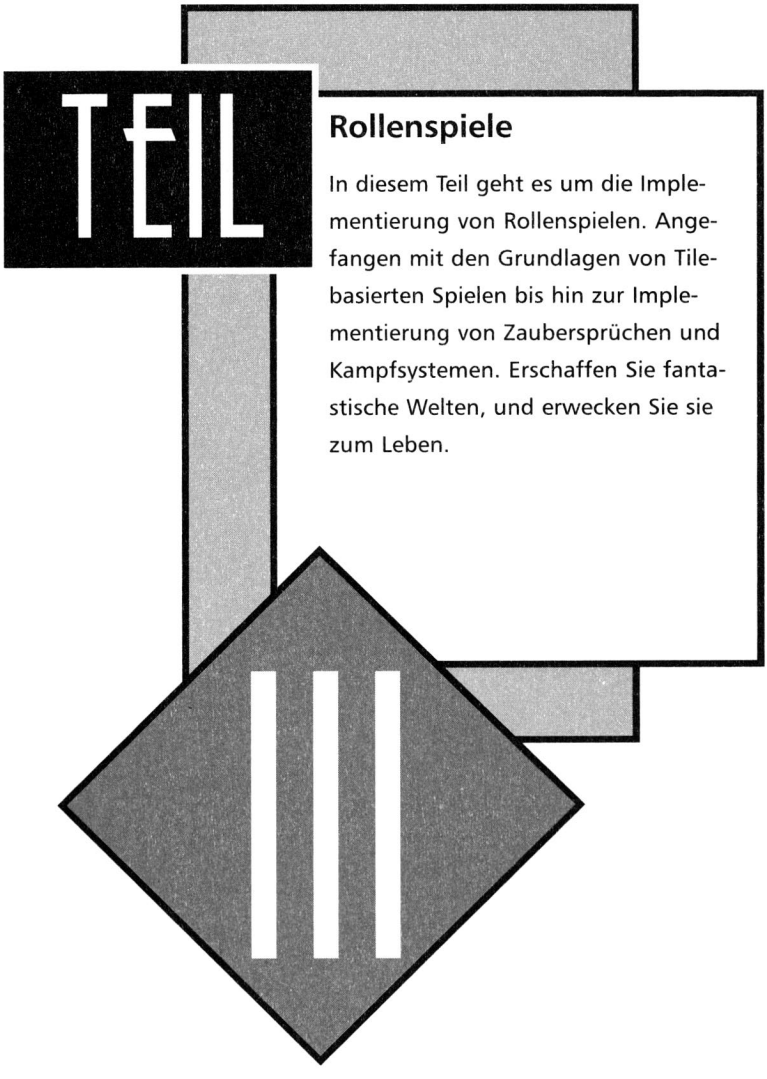
Die meisten Spielezeitschriften haben ein eigenes Nachrichtenboard. Schreiben Sie einen Hinweis auf diese Schwarzen Bretter, und nehmen Sie sich die Zeit auf mögliche Kommentare zu antworten.

Schreiben Sie auch die Redaktionen dieser Zeitschriften direkt an, eventuell bringen Sie ja eine kurze Notiz oder veröffentlichen Ihr Spiel sogar auf der Cover-CD.

Der Fernsehsender NBC-Giga hat auf seiner Webseite einen »Free and Fun«-Bereich. Auch hierhin sollten Sie eine Email versenden und im entsprechenden Forum eine Nachricht hinterlassen.

Und natürlich sollten Sie gerade an die Webseiten der von Ihnen erkannten Zielgruppe schreiben. Und auch hier ist eine Präsenz in den Foren sehr wichtig.

Wenn das jetzt beinahe nach mehr Arbeit klingt als das eigentliche Schreiben des Spiels, dann liegen Sie richtig. Aber denken Sie daran: Wenn keiner Ihr Spiel kennt, dann wird es auch niemand spielen.



Rollenspiele

In diesem Teil geht es um die Implementierung von Rollenspielen. Angefangen mit den Grundlagen von Tile-basierten Spielen bis hin zur Implementierung von Zaubersprüchen und Kampfsystemen. Erschaffen Sie fantastische Welten, und erwecken Sie sie zum Leben.

23 Tile Based Maps

Rollenspiele haben teilweise riesige Gebiete, die es zu erforschen gilt. Aus diesem Grund benutzen Sie ein sehr praktisches System, um große Karten platzsparend zu speichern: *Tile Based Maps*.

Rogue

Rollenspiele haben einen langen Weg hinter sich. Im Jahre 1980 haben die Studenten Glenn Wichmann, Ken Arnold und Michael Toy ihr Spiel »Rogue« veröffentlicht. In diesem Spiel begab sich ein Abenteurer in die »Kerker des Unheils«, um Monster zu besiegen und am Ende mit dem Amulett von Yendor wieder an die Oberfläche zu kommen. Das Besondere an diesem Spiel waren die zufällig erzeugten Verliese, die jedes Spiel unterschiedlich machten. Auf jeder Ebene gab es eine Treppe, die zu dem nächst tieferen Verlies führte. Je tiefer man in die Verliese vordrang, umso gefährlicher wurde es. Es gab Fallen, Geheimtüren, Schätze und natürlich jede Menge Monster, die den Spieler angriffen, sobald sie ihn sehen konnten. Der Spieler konnte seine anfangs recht spärliche Ausrüstung durch Gegenstände ersetzen, die er im Verlies fand. Einige der Monster hinterließen einen Schatz, wenn sie besiegt wurden.

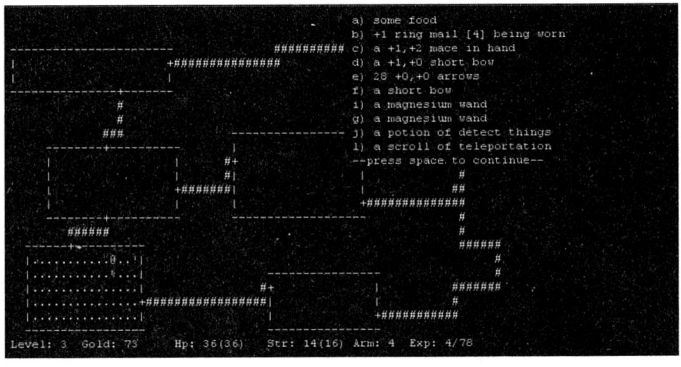


Abbildung 23.1: Rogue

Anstatt Grafik nutzte Rogue ASCII-Zeichen, um die Verliese auf dem Bildschirm darzustellen. Dies machte es möglich, große Karten (englisch: Maps) in dem damals sehr stark begrenzten Speicher zu verwalten, da jede Position durch ein einzelnes Byte repräsentiert werden konnte.

Kacheln

Allerdings sehen natürlich die Buchstaben, Satz- und Sonderzeichen nicht gerade umwerfend aus. Als dann die PCs langsam grafikfähig wurden, kam jemand auf die Idee, anstatt der Zeichen kleine Grafiken zu benutzen. Das Spielfeld wurde also immer noch in Felder aufgeteilt, und jedes Feld wurde durch ein Byte repräsentiert. Aber anstatt des ASCII-Codes eines Zeichens wurde nun der Index einer Grafik gespeichert. Diese Grafiken werden als *Tiles* (englisch für »Kachel«) bezeichnet.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	0
0	0	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	1	0
0	0	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	1	0
0	0	1	6	6	6	6	6	8	6	6	6	6	6	6	6	6	6	1	0
0	0	1	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	1	0
0	0	4	1	1	1	3	6	6	6	6	6	6	6	6	6	6	6	1	0
0	0	0	0	0	0	1	6	6	6	6	6	6	6	6	6	6	6	1	0
0	0	0	0	0	0	4	1	1	1	1	1	1	1	1	1	1	1	5	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 23.2: Die Werte in einer Map

Jedes Byte steht für eine bestimmte Grafik. Und so wird aus dem wirren Zahlensalat die Karte eines Verlieses.

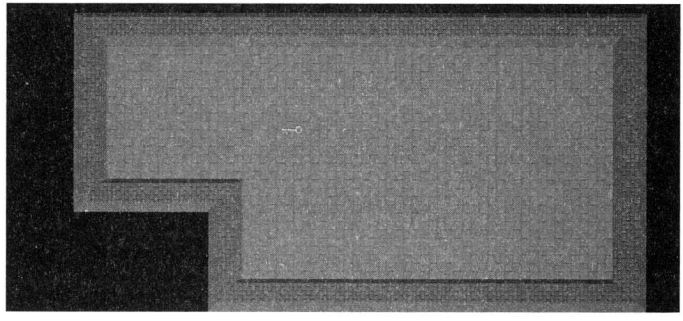


Abbildung 23.3: Die gleiche Map mit Tiles gerendert

Bei einer Tilegröße von 32x32 Pixel und einer Bildschirmauflösung von 640x480 Pixel füllt eine Karte mit 20x15 Tiles den kompletten Bildschirm.

Vorerst benutzen wir zum Definieren der Karte einen festen String innerhalb des Programms.

```
const int MAP_W  = 20;
const int MAP_H  = 15;
const int TILE_W = 32;
const int TILE_H = 32;

char tileMapStr[MAP_W * MAP_H+1] =
//0...4....90...4....9
 "1---------------2   " // 0
 "!...............!   " // 1
 "!...............!   " // 2
 "!...............3--2" // 3
 "!..................!" // 4
 "3----2.............!" // 5
 "     !.............!" // 6
 "     !.............!" // 7
 "     !.............!" // 8
 "     !.............!" // 9
 "  1--4.............!" // 10
 "  !................!" // 11
 "  !................!" // 12
 "  3-----2..........!" // 13
 "         3---------4" // 14
;
```

Jedes Zeichen in `tileMapStr` steht für ein Tile. Die Zuordnung von Zeichen zu Tile Index wird über einen weiteren String vorgenommen.

```
char char2tile[] =" -!1234.k";
```

Die Position der Zeichen in diesem String entspricht dem Tile Index. Das Leerzeichen entspricht also Tile Index 0, der Bindestrich ist Index 1 und so weiter. Allerdings wäre es recht langsam, beim Zeichnen der Map immer den `char2tile`-String zu durchsuchen, um den Index der korrekten Kachel zu finden. Aus diesem Grund wird der `tileMapStr` zu Beginn des Programms umgeformt, um direkt den Tile Index zu speichern.

Nachdem der String umgeformt wurde, kann er nicht mehr mit den normalen Textfunktionen angezeigt werden, da jedes Leerzeichen zu einem '\0' wird, welches das Stringende in C/C++ repräsentiert. Aus diesem Grund ist es auch notwendig, die Größe der Karte, also des Strings zu kennen, da keine normale Stringende-Kennung mehr vorhanden ist.

Das Umwandeln erledigt die Funktion `setupMap()`:

```
void setupMap(char *map, int size) {
    for (int a=0; a < size; a++) {
        for (int tile=0; char2tile[tile]; ++tile) {
            if (map[a] == char2tile[tile]) {
                map[a] = tile;
                break;
            }
        }
    }
}
```

Diese Funktion geht nach einem recht einfachen Schema vor. Der übergebene String `map` wird Zeichen für Zeichen durchsucht. Die Variable `a` ist die Position innerhalb der Map. Das Zeichen `map[a]` ist demzufolge das derzeitig untersuchte Zeichen. In der inneren Schleife wird `char2tile` Zeichen für Zeichen mit `map[a]` verglichen. Wird eine Übereinstimmung gefunden, dann wird `map[a]` auf die Position gesetzt, an der das Zeichen in `char2tile` gefunden wurde.

Da nun `tileMapStr` nur noch den direkten Index auf das jeweilige Tile enthält, kann man die Karte auf sehr einfache Weise anzeigen:

```
void drawMap(char* map, BITMAP *tiles) {
    int pos = 0;
    for (int y=0; y < MAP_H; ++y) {
        for (int x=0; x < MAP_W; ++x) {
            drawTile(doubleBuffer, tiles,
                     map[pos], x*TILE_W, y*TILE_H);
            ++pos;
        }
    }
}
```

Die Funktion `drawMap` erwartet als Parameter die Karte (`map`) und die Bitmap mit den Kacheln (`tiles`). Dann wird die Karte Zeile für Zeile durchgegangen und das entsprechende Tile wird angezeigt.

Wenn die `tiles`-Bitmap alle Kacheln nacheinander enthält (also Kachel 0 an Position (0,0), Kachel 1 and Position (32,0) und so weiter), dann kann man ein einzelnes Tile durch einen einzigen Aufruf von `blit()` anzeigen.

```
void drawTile(BITMAP *dest, BITMAP *tile, int index,
                                        int x, int y) {
    blit(tile, dest,
         index * TILE_W, 0, x, y, TILE_W, TILE_H
        );
}
```

Je nachdem, welchen Satz von Kacheln man benutzt, kann das Ergebnis sehr unterschiedlich wirken. Benutzt man Tiles, die wie Steinblöcke aussehen, dann erhält man das typische Verlies (siehe Abbildungen 23.4 und 23.5).

Abbildung 23.4: Ein Tile Set aus Steinen

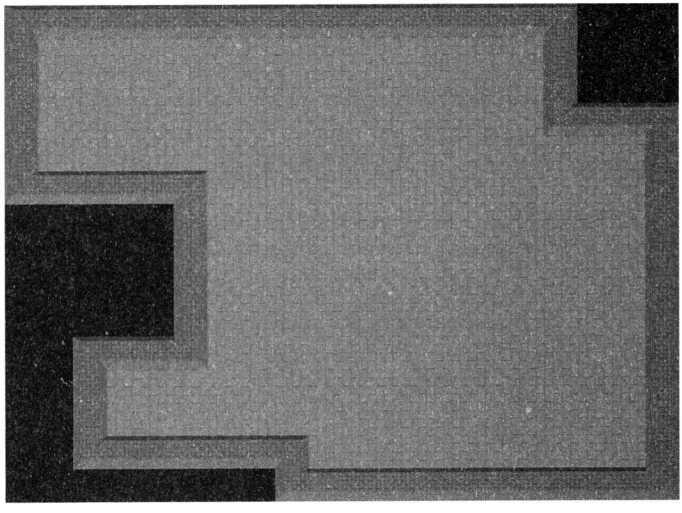


Abbildung 23.5: Screenshot tiles0.exe mit Stein-Tile-Set

Wenn man jedoch ein anderes Tile Set benutzt, dann sieht die gleiche Karte auf einmal ganz anders aus.

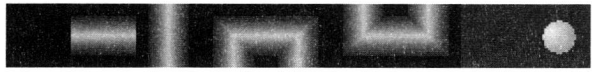


Abbildung 23.6: Ein alternatives Tile Set

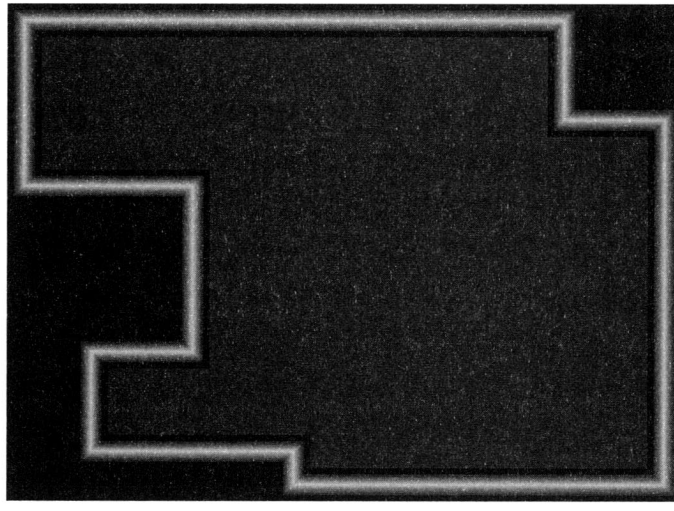


Abbildung 23.7: Die Karte mit dem alternativen Satz an Kacheln

Durch die Verwendung des blauen Neon-Tile-Sets wirkt die Karte auf einmal deutlich kühler und vermittelt durch den Farbverlauf in den Kacheln einen Hightech-Look. So könnten Sie zum Beispiel die Handlung von einer Fantasywelt in den Cyberspace verlagern, ohne den Quellcode verändern zu müssen.

Den kompletten Quellcode für dieses Beispiel (Tiles0.cpp) finden Sie auf der CD, im Quellcode-Verzeichnis dieses Kapitels.

Die Map-Klasse

Im letzten Beispiel haben wir eine globale Konstante benutzt, um die Größe der Karten zu speichern. Dies hat einige Nachteile. Zum einen legen wir damit die Größe aller Karten unwiderruflich fest, zum anderen trennen wir damit zusammengehörige Daten.

Die Karte, deren Größe und die Tiles, die zum Anzeigen benötigt werden, bilden zusammen mit den entsprechenden Methoden eine logische Einheit.

```
// Map.h
#ifndef MAP_HEADER
#define MAP_HEADER

#include <allegro.h>

class Map {
private:
    int     w, h;
    char*   data;
    BITMAP *tiles;
    int     tileWidth;
    int     tileHeight;

    inline void drawTile(BITMAP *dest, int x, int y, int tile);
    void        create(int w, int h);

public:
    Map() : w(0), h(0), data(NULL), tiles(NULL) {
    }
    Map(int w, int h);
    Map(int w, int h, char* mapStr, char* tileStr);
    virtual ~Map();

    void setTiles(BITMAP *tiles, int tw, int th);
    void draw(BITMAP *dest);
};

#endif
```

Die Implementierung dieser Methoden bleibt beinahe wie gehabt, nur überschreibt der Konstruktor nun nicht mehr den übergebenen String,

sondern legt ein neues Array (data) an, und speichert dann die angepasste Form der Map in diesem Array. Neu hinzugekommen sind auch die Methoden, um ein Tile Set zu setzen. Ansonsten ist aber alles beim Alten geblieben.

```
// Map.cpp
#include "map.h"

void Map::drawTile(BITMAP *dest, int x, int y, int tile) {
    blit(tiles, dest, tile * tileWidth, 0, x, y, tileWidth,
    tileHeight);
}

void Map::create(int w, int h) {
    this->w = w;
    this->h = h;
    this->data = new char[w*h];
}

Map::Map(int w, int h) {
    create(w, h);
}

Map::Map(int w, int h, char* mapStr, char* tileStr) {
    create(w, h);
    int size = w* h;
    for (int a=0; a < size; ++a) {
        for (int tile=0; tileStr[tile]; ++tile) {
            if (mapStr[a] == tileStr[tile]) {
                data[a] = tile;
                break;
            }
        }
    }
}

Map::~Map() {
    if (data) {
        delete [] data;
    }
}

void Map::setTiles(BITMAP *tiles, int tw, int th) {
    this->tiles = tiles;
```

```
    tileWidth   = tw;
    tileHeight  = th;
}

void Map::draw(BITMAP *dest) {
    int pos = 0;
    for (int y=0; y < h; ++y) {
        for (int x=0; x < w; ++x) {
            drawTile(dest, x*tileWidth, y*tileHeight, data[pos]);
            ++pos;
        }
    }
}
```

Die Verwendung der Map-Klasse lässt auch den Umfang des Beispielprogramms merklich schrumpfen und erhöht somit die Wartbarkeit des Quellcodes.

```
#include <allegro.h>
#include "util.h"

#include "map.h"

const int MAP_W  = 20;
const int MAP_H  = 15;
const int TILE_W = 32;
const int TILE_H = 32;


char tileMapStr[MAP_W * MAP_H+1] =
//0...4....90...4....9
 "1--------------2   " // 0
 "!..............!   " // 1
 "!..............!   " // 2
 "!..............3--2" // 3
 "!................!" // 4
 "3----2............!" // 5
 "     !............!" // 6
 "     !............!" // 7
 "     !............!" // 8
 "     !............!" // 9
 "  1--4............!" // 10
 "  !...............!" // 11
 "  !...............!" // 12
 "  3----2..........!" // 13
 "       3----------4" // 14
```

```
;
char char2tile[] =" -!1234.k";


int main(int , char**) {
    init(640, 480, 60);

    BITMAP *tiles = load_bitmap("tiles1.tga", NULL);

    Map map(MAP_W, MAP_H, tileMapStr, char2tile);
    map.setTiles(tiles, TILE_W, TILE_H);
    map.draw(doubleBuffer);

    show();
    clear_keybuf();
    readkey();

    destroy_bitmap(tiles);
    return 0;
} END_OF_MAIN()
```

Zusammenfassung

Die Idee von Tile Based Maps ist es, eine komplexe Grafik (die Welt, in der sich der Spieler bewegt) aus vielen kleinen, gleich großen Grafiken zusammenzusetzen. Diese Grafiken nennt man Tiles (Kacheln). Jedes Tile hat eine bestimmte Nummer, den Index. Die Spielwelt wird nun in Felder von der Größe der Kacheln aufgeteilt. Die dabei entstehende Karte bezeichnet man als Tile Map. Jedes Feld speichert genau einen Index. Beim Anzeigen wird der Index aus der Tile Map ausgelesen, und die entsprechende Grafik angezeigt.

Übungen

Es ist wichtig, dass Sie das Prinzip der Kacheln und Karten verstehen. Aus diesem Grund sollten Sie einige Zeit mit dem Beispielprogramm `tiles1.cpp` experimentieren. Ändern Sie den `tileMapStr`, fügen Sie neue Kacheln zu der Grafik hinzu und geben Sie zu der Grafik auch den Index aus (mit `textprintf()`).

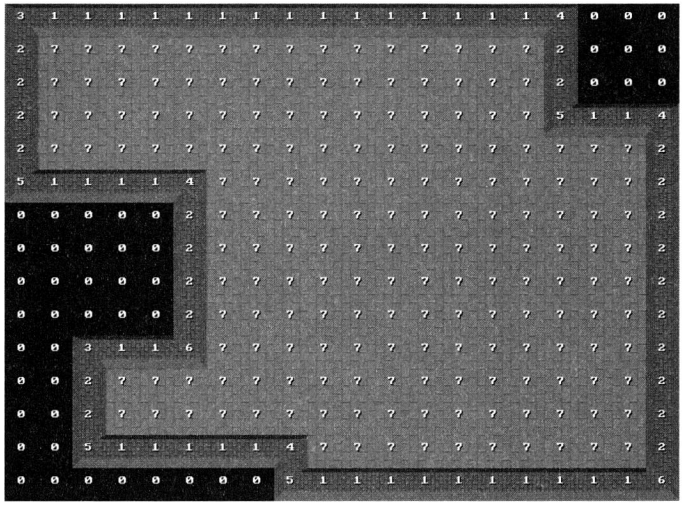


Abbildung 23.8: Karte mit angezeigtem Tile Index

24 Anzeigen von Spielfiguren

Im letzten Kapitel haben wir die Karte eines Verlieses angezeigt. Karten von Verliesen sind ohne Leben jedoch nicht gerade spannend. In diesem Kapitel werden wir nun eine Spielfigur anzeigen und dafür sorgen, dass diese Figur nicht durch die Wände des Verlieses laufen kann.

Sprites in einer Welt aus Kacheln

Unser Held soll sich in allen vier Richtungen über die Map bewegen können. In der Draufsicht könnten wir das Sprite in nur einer Richtung zeichnen und dann je nach gewünschter Richtung drehen. Allerdings funktioniert dies wirklich nur in der Draufsicht. Aus diesem Grund benutzen wir hier eine andere Methode, die auch in der Klapp-Perspektive funktioniert. Dies erlaubt es uns, die Perspektive zu wechseln (durch die Verwendung anderer Tilegrafiken), ohne den Quellcode ändern zu müssen.

Der Nachteil ist, dass wir für das Heldensprite ein paar Frames mehr brauchen.

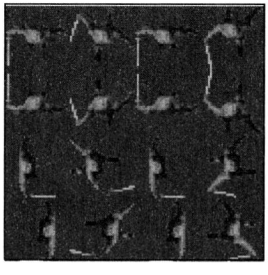


Abbildung 24.1: Der Held (32x32) in vier Richtungen

Wie Sie sehen ist die Reihenfolge der Frames immer die gleiche. Wenn Sie also beim 2ten Frame der »Nach Rechts«-Animation sind und den Charakter nach oben bewegen, dann können Sie einfach zum 2ten Frame der »Nach Oben«-Animation wechseln. In der Tat können so die Animation und die Richtung komplett getrennt voneinander behandelt werden.

Die Sprite-Klasse

Die Sprite-Klasse muss in erster Linie dafür sorgen, dass sich das Sprite mit einer angemessenen Geschwindigkeit bewegt und sicher stellen, dass immer der korrekte Frame für die derzeitige Richtung angezeigt wird.

Wir brauchen also die Position des Sprites auf dem Schirm, die Richtung, in die der Held blickt, den aktuellen Frame der Animation und die Animationsgeschwindigkeit.

Damit das Sprite sich auch bewegen kann, braucht es eine `update()`-Methode und für die Anzeige eine `draw()`-Methode.

Um die Geschwindigkeit des Sprites möglichst flexibel verändern zu können (schließlich sollen sich ja später verschiedene Figuren unterschiedlich schnell bewegen können), wird ein `updateCounter` benutzt. Dieser wird bei jedem Aufruf von `update()` um den Wert der `speed`-Variable erhöht. Sobald der `updateCounter` den Wert von 1.0 erreicht, wird das Sprite bewegt und eine neuer Frame angezeigt.

Erhöht man die Geschwindigkeit, indem man die `speed`-Variable zum Beispiel von 0.1 auf 0.3 erhöht, dann bewegt sich das Sprite sofort um den Faktor 3 schneller.

In Verbindung mit `deltaX`- und `deltaY`-Werten, die für jeden Frame gesetzt werden können, haben Sie also die Möglichkeit, sowohl die Weite eines jeden Schritts als auch die Geschwindigkeit der Schritte nach Belieben einzustellen.

```
#ifndef SPRITE_HEADER
#define SPRITE_HEADER

#include <allegro.h>
#include "util.h"

class Sprite {
    int frameWidth;
    int frameHeight;
    int curFrame;
    int curDirection;
    bool moving;

    BITMAP *frames;
    int countFrames;
```

```
    int *deltaX;
    int *deltaY;

    int x;
    int y;

    float speed;
    float updateCounter;

    SAMPLE *stepSnd;
public:
    enum {
        DOWN  = 0,
        UP    = 1,
        RIGHT = 2,
        LEFT  = 3
    };
    Sprite() : frameWidth(0), frameHeight(0),
               curFrame(0)  , curDirection(0),
               frames(NULL) , countFrames(0),
               deltaX(NULL) , deltaY(NULL),
               x(0)         , y(0),
               speed(0.1)   , updateCounter(0.0),
               stepSnd(NULL)                     {
    }

    Sprite(BITMAP *img, int fw, int fh);

    ~Sprite();

    void setMoving(bool move);
    bool isMoving();
    void setDirection(int dir);

    void setX(int x);
    void setY(int y);
    void setPosition(int x, int y);

    int  getX();
    int  getY();

    void setDeltaX(int frame, int delta);
    void setDeltaY(int frame, int delta);

    void setSpeed(float speed);
    void setStepSound(SAMPLE *snd);
```

```
    void update();
    void draw(BITMAP *dest);
};

#endif
```

Und hier auch gleich ohne weitere Umschweife der Quellcode für die Implementierung der Sprite-Klasse:

```
#include "sprite.h"

Sprite::Sprite(BITMAP *img, int fw, int fh) {
    frames      = img;
    frameWidth  = fw;
    frameHeight = fh;
    countFrames = img->w/ fw;

    curFrame = curDirection = 0;
    moving   = false;

    speed = 0.1;

    deltaX = new int[countFrames];
    deltaY = new int[countFrames];

    for (int a=0; a < countFrames; ++a) {
        deltaX[a] = deltaY[a] = 0;
    }

    stepSnd = NULL;
}

Sprite::~Sprite() {
    if (!deltaX) {
        delete [] deltaX;
        deltaX = NULL;
    }
    if (!deltaY) {
        delete [] deltaY;
        deltaY = NULL;
    }
}

void Sprite::setMoving(bool move) {
    if (!move) {
```

```
        curFrame = 0;
    }
    moving = move;
}
void Sprite::setDirection(int dir) {
    curDirection = dir;
}

bool Sprite::isMoving() {
    return moving;
}

void Sprite::setX(int x) {
    this->x = x;
}
void Sprite::setY(int y) {
    this->y = y;
}

void Sprite::setPosition(int x, int y) {
    this->x = x;
    this->y = y;
}

int Sprite::getX() {
    return x;
}

int Sprite::getY() {
    return y;
}

void Sprite::setDeltaX(int frame, int delta) {
    deltaX[frame] = delta;
}

void Sprite::setDeltaY(int frame, int delta) {
    deltaY[frame] = delta;
}

void Sprite::setSpeed(float speed) {
    this->speed = speed;
}
```

```
void Sprite::setStepSound(SAMPLE *snd) {
    stepSnd = snd;
}

void Sprite::update() {
    if (isMoving()) {
        updateCounter += speed;
        while (updateCounter >= 1.0) {
            curFrame++;
            if (curFrame >= countFrames) {
                curFrame -= countFrames;
            }
            if (stepSnd && (curFrame &1)==1) {
                playSound(stepSnd, 128);
            }
            switch (curDirection) {
                case DOWN:
                    x += deltaX[curFrame];
                    y += deltaY[curFrame];
                break;

                case UP:
                    // Y negieren
                    x += deltaX[curFrame];
                    y -= deltaY[curFrame];
                break;

                case RIGHT:
                    // X und Y vertauschen
                    x += deltaY[curFrame];
                    y += deltaX[curFrame];
                break;

                case LEFT:
                    // X und Y vertauschen
                    // X negieren
                    x -= deltaY[curFrame];
                    y += deltaX[curFrame];
                break;
            }
            updateCounter -= 1.0;
        }
    } else {
        updateCounter = 0.0;
    }
```

```
}

void Sprite::draw(BITMAP *dest) {
    if (!frames) {
        return;
    }
    masked_blit(frames, dest,
                curFrame * frameWidth,
                curDirection * frameHeight,
                x, y,
                frameWidth, frameHeight);
}
```

Die `update()`-Methode ist das Kernstück der Sprite-Klasse. Zuerst wird überprüft ob sich das Sprite in Bewegung befindet. Ist dies der Fall, wird der `updateCounter` um `speed` erhöht. Findet keine Bewegung statt, dann wird der `updateCounter` wieder auf 0.0 gesetzt.

Wenn der Wert von `updateCounter` 1.0 übersteigt, dann wird zum nächsten Frame gewechselt, und die Position entsprechend des `deltaX`, `deltaY`-Arrays angepasst.

Da nur die Delta-Werte für eine Richtung (`Sprite::DOWN`) gesetzt werden, ist es notwendig, bei den anderen Richtungen entweder das Vorzeichen zu wechseln oder `deltaX` und `deltaY` zu vertauschen. Sie können sich vorstellen, dass der `deltaY`-Wert die Bewegung in Laufrichtung angibt, und der `deltaX`-Wert die Bewegung senkrecht zur Laufrichtung.

Um bei jedem Schritt auch ein passendes Geräusch abzuspielen, wurde `util.cpp` um die `playSound()`-Funktion aus Kapitel 20 erweitert:

```
void playSound(SAMPLE *sample, int vol) {
    int freq= rnd(100) + 950;

    vol *= 5+(rnd(5)+1);
    vol /= 10;
    play_sample(sample, vol, 128, freq, 0);
}
```

Der Sound wird nur bei ungeraden Frames abgespielt (`curFrame &1` ist nur dann ungleich 0, wenn die 1er Stelle im Wert gesetzt ist. Und ist diese gesetzt, dann muss die Zahl ungerade sein.).

Wenn wir nun Tile Map Code und Sprite Code kombinieren, dann erhalten wir einen durch den Kerker laufenden Barbaren.

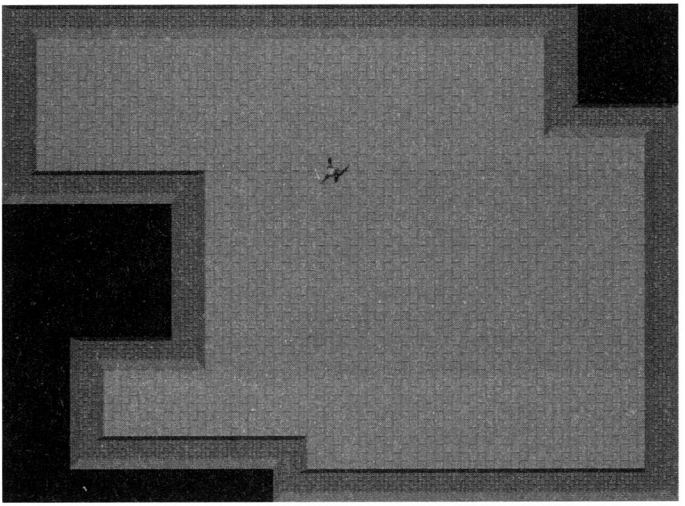


Abbildung 24.2: Kerker mit Held

Die Abfrage des Joysticks/der Tastatur übernimmt die in Kapitel 12 entwickelte Joypad-Klasse.

Die einzigen zusätzlichen Abfragen dienen dem Festlegen der Geschwindigkeit. Die Tasten 1 bis 6 erlauben es Ihnen die Geschwindigkeit innerhalb des Bereichs von 0.1 bis 0.2 zu verschieben.

Mit Esc verlassen Sie das Beispielprogramm.

```
#include <allegro.h>
#include "util.h"

#include "map.h"
#include "sprite.h"
#include "joypad.h"

const int MAP_W  = 20;
const int MAP_H  = 15;
const int TILE_W = 32;
const int TILE_H = 32;
```

```
char tileMapStr[MAP_W * MAP_H+1] =
//0...4....90...4....9
 "1---------------2   " // 0
 "!...............!   " // 1
 "!...............!   " // 2
 "!...............3--2" // 3
 "!..................!" // 4
 "3----2.............!" // 5
 "     !.............!" // 6
 "     !.............!" // 7
 "     !.............!" // 8
 "     !.............!" // 9
 "  1--4.............!" // 10
 "  !................!" // 11
 "  !................!" // 12
 "  3-----2..........!" // 13
 "        3----------4" // 14
;
char char2tile[] =" -!1234.k";



Joypad *joypad;

int main(int , char**) {
    init(640, 480, 60);

    joypad = new Joypad();

    BITMAP *tiles   = load_bitmap("tiles0.tga", NULL);
    BITMAP *heroSpr = load_bitmap("hero.tga"  , NULL);

    SAMPLE *footStep = load_sample("step.wav");

    Sprite *hero = new Sprite(heroSpr, TILE_W, TILE_H);

    for (int a=0; a < 4; ++a) {
        hero->setDeltaY(a, 8);
    }
    hero->setPosition(SCREEN_W/2, SCREEN_H/2);
    hero->setStepSound(footStep);

    Map map(MAP_W, MAP_H, tileMapStr, char2tile);
    map.setTiles(tiles, TILE_W, TILE_H);
```

```
bool needsRedraw = false;

timerCounter = 0;
syncTimer(&timerCounter);
while (!key[KEY_ESC]) {
    if (timerCounter) {
        joypad->poll();
        if (joypad->button[Joypad::UP]) {
            hero->setDirection(Sprite::UP);
            hero->setMoving(true);
        } else if (joypad->button[Joypad::DOWN]) {
            hero->setDirection(Sprite::DOWN);
            hero->setMoving(true);
        } else if (joypad->button[Joypad::LEFT]) {
            hero->setDirection(Sprite::LEFT);
            hero->setMoving(true);
        } else if (joypad->button[Joypad::RIGHT]) {
            hero->setDirection(Sprite::RIGHT);
            hero->setMoving(true);
        } else {
            hero->setMoving(false);
        }

        if (key[KEY_1]) {
            hero->setSpeed(0.1);
        } else if (key[KEY_2]) {
            hero->setSpeed(0.12);
        } else if (key[KEY_3]) {
            hero->setSpeed(0.14);
        } else if (key[KEY_4]) {
            hero->setSpeed(0.16);
        } else if (key[KEY_5]) {
            hero->setSpeed(0.18);
        } else if (key[KEY_6]) {
            hero->setSpeed(0.2);
        }

        do {
            hero->update();
            --timerCounter;
        } while (timerCounter >0);
        needsRedraw = true;
    }

    if (needsRedraw) {
```

```
            map.draw(doubleBuffer);
            hero->draw(doubleBuffer);

            needsRedraw = false;
            show();
        }
    }

    delete hero;
    destroy_bitmap(heroSpr);
    destroy_bitmap(tiles);
    return 0;
} END_OF_MAIN()
```

Den kompletten Quellcode finden Sie wie immer auf der CD.

Sieht doch schon recht gut aus. Es gibt nur ein Problem: Das Sprite ignoriert seine Umgebung komplett. Sie können ohne jedes Problem durch Wände gehen und den sichtbaren Bereich verlassen. Aber darum kümmern wir uns sofort.

Nicht mit dem Kopf durch die Wand

Damit unser Sprite erkennen kann, wo die Wände sind, muss es wissen, in welcher `Map` es sich befindet. Solange die `Sprite`-Klasse keine Referenz auf die Karte hat, kann es nicht abfragen, ob der Bereich begehbar ist oder nicht.

Also geben wir dem Sprite einen Zeiger auf die Karte mit. Dann kann das Sprite die Map fragen, ob es eine bestimmte Stelle betreten kann. Das heißt: Das Sprite könnte die Map fragen, wenn diese entsprechende Funktionen hätte. Erweitern wir die `Map`-Klasse um Methoden, ein einzelnes Tile abzufragen und zu setzen. Und damit die `Sprite`-Klasse kein tieferes Wissen über die Karte benötigt, spendieren wir der `Map`-Klasse noch eine Methode, um abzufragen, ob ein bestimmtes Feld betreten werden kann.

Da die Zugriffsfunktionen die Größe des `data`-Arrays benötigen, legen wir für die Größe eine weitere Variable an und setzen diese in der `create()`-Methode der `Map`.

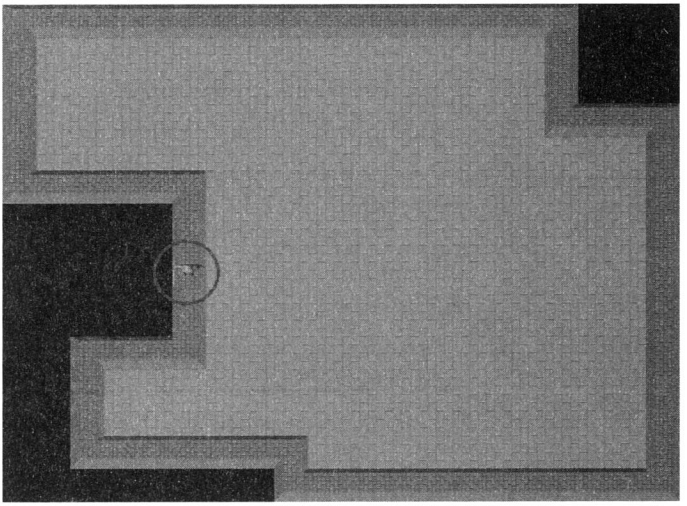


Abbildung 24.3: Der Held, wo er eigentlich nicht sein dürfte

```
void Map::create(int w, int h) {
    this->w = w;
    this->h = h;
    this->size = w*h;
    this->data = new char[size];
}

int  Map::getTileAt(int tileX, int tileY) {
    int pos = tileX + tileY * w;
    if (pos >=0 && pos < size) {
        return data[pos];
    }
    return -1;
}

void Map::setTileAt(int tileX, int tileY, int tile) {
    int pos = tileX + tileY * w;
    if (pos >=0 && pos < size) {
        data[pos] = (char) (tile & 0xff);
    }
}
```

```
bool Map::isWalkable(int tileX, int tileY) {
    int tile = getTileAt(tileX, tileY);
    return (tile >=7);
}
```

Die `isWalkable()`-Methode nutzt in diesem Beispiel noch Wissen, welches die Karte eigentlich nicht haben darf. Wir müssen uns bald eine Möglichkeit überlegen, die Begehbarkeit eines bestimmten Feldes festzulegen, ohne auf inneres Wissen über die Kachel zurückgreifen zu müssen.

Doch für den Augenblick ist diese Methode gut genug. Jetzt muss die `update()`-Methode des Sprites angepasst werden.

Bevor wir die Position in Abhängigkeit von der Richtung ändern, speichern wir die aktuelle Position in zwei temporären Variablen.

Nachdem die Position angepasst wurde, nutzen wir die Methoden der `Map` Klasse, um die Position des Sprites in eine Feldposition umzuwandeln. Dann kann die `isWalkable()`-Methode benutzt werden, um die Begehbarkeit des Feldes zu testen. Wenn das Feld nicht begehbar ist, dann wird die Position zurückgesetzt.

```
void Sprite::update() {
    if (isMoving()) {
        updateCounter += speed;
        while (updateCounter >= 1.0) {
            curFrame++;
            if (curFrame >= countFrames) {
                curFrame -= countFrames;
            }
            if (stepSnd && (curFrame &1)==1) {
                playSound(stepSnd, 128);
            }
            int ox = x;
            int oy = y;
            switch (curDirection) {
                case DOWN:
                    x += deltaX[curFrame];
                    y += deltaY[curFrame];
                break;

                case UP:
                    // Y negieren
                    x += deltaX[curFrame];
                    y -= deltaY[curFrame];
                break;
```

```
                case RIGHT:
                    // X und Y vertauschen
                    x += deltaY[curFrame];
                    y += deltaX[curFrame];
                break;

                case LEFT:
                    // X und Y vertauschen
                    // X negieren
                    x -= deltaY[curFrame];
                    y += deltaX[curFrame];
                break;
            }
            if (map) {
                int tw = map->getTileWidth();
                int th = map->getTileHeight();

                int tx = x / tw;
                int ty = y / th;

                if (!map->isWalkable(tx, ty)) {
                    x = ox;
                    y = oy;
                }
            }
            updateCounter -= 1.0;
        }
    } else {
        updateCounter = 0.0;
    }
}
```

Nun müssen Sie noch im Hauptprogramm die `setMap()`-Methode des Sprites aufrufen.

```
Map map(MAP_W, MAP_H, tileMapStr, char2tile);
map.setTiles(tiles, TILE_W, TILE_H);

hero->setMap(&map);
```

Und nun rennt das Sprite auch nicht mehr kopflos in Wände.

Den vollständigen Quellcode für dieses Beispiel finden sie auf der beiliegenden CD.

25 Komplexere Karten

Ich habe ja bereits einige Male geschrieben, dass einer der Pluspunkte einer Tile Map die Speicherplatzgewinne bei großen Karten ist. Leider konnten wir diesen Vorteil bisher nicht wirklich ausnutzen, da alle Karten auf die Größe des Schirmes beschränkt waren. Doch dies werden wir nun ändern.

In diesem Kapitel werden wir uns mit dem Scrolling der Karte beschäftigen und für jedes Feld Attribute setzten. Zum Abschluss erweitern wir die Karte so, dass mehrere Kacheln übereinander liegen können.

Größere Karten

Bisher haben wir die Karte immer komplett angezeigt und dabei das obere linke Tile immer auch an der oberen linken Ecke der übergebenen Bitmap platziert.

```
void Map::draw(BITMAP *dest) {
    int pos = 0;
    for (int y=0; y < h; ++y) {
        for (int x=0; x < w; ++x) {
            drawTile(dest,
                     x*tileWidth, y*tileHeight,
                     data[pos]);
            ++pos;
        }
    }
}
```

Dies ist eine sehr einfache, aber auch sehr inflexible Lösung. Sollte die übergebene `dest`-Bitmap zum Beispiel kleiner sein als `(tileWidth*w, tileHeight*h)`, dann würde unnötig Zeit damit verschwendet, die Bereiche der Karte zu zeichnen, die nicht sichtbar sind (siehe Abbildung 25.1).

Die Anzahl der Tiles, die tatsächlich angezeigt werden können, lässt sich direkt aus der Größe eines Tiles und der Größe der Zielbitmap berechnen.

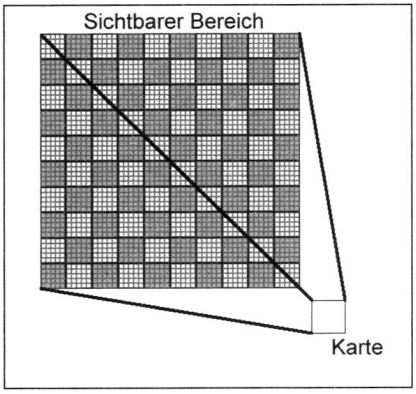


Abbildung 25.1: Zusammenhang sichtbarer Bereich und Karte bisher

```
void Map::draw(BITMAP *dest) {
    int pos = 0;

    int w = MIN(this->w, dest->w / tileWidth);
    int h = MIN(this->h, dest->h / tileHeight);

    for (int y=0; y < h; ++y) {
        for (int x=0; x < w; ++x) {
            drawTile(dest,
                     x*tileWidth, y*tileHeight,
                     data[pos]);
            ++pos;
        }
    }
}
```

Das Problem hier ist jedoch, dass so teilweise eine Kachel zu wenig angezeigt wird. Ist die Zielbitmap zum Beispiel 340 Pixel breit, dann wird die Anzahl der Kacheln falsch berechnet (340 / 32 = 10; da bei einer Integer-Division der Nachkommateil ignoriert wird).

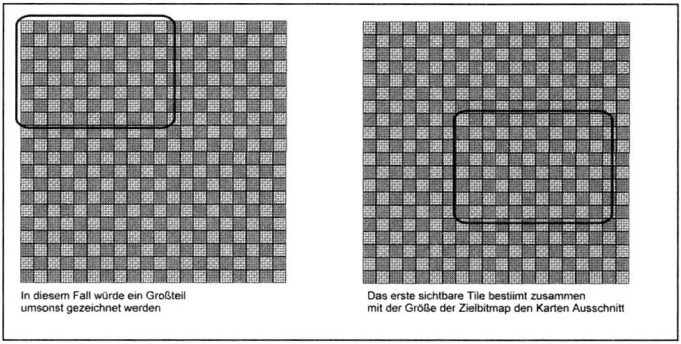


Abbildung 25.2: Größere Karten im Überblick

Es ist also notwendig, die Anzahl der Tiles um 1 zu erhöhen, wenn der Nachkommateil ungleich 0 ist.

```
void Map::draw(BITMAP *dest) {
    int pos = 0;

    int w = MIN(this->w, dest->w / tileWidth
                + dest->w % tileWidth  == 0 ? 0 : 1);
    int h = MIN(this->h, dest->h / tileHeight
                + dest->h % tileHeight == 0 ? 0 : 1);
    // …
}
```

Jetzt wird die Anzahl der sichtbaren Kacheln korrekt berechnet. Allerdings wird jetzt die Anzahl der sichtbaren Tiles jedes Mal neu berechnet. Wir können davon ausgehen, dass die Zielbitmap in den meisten Fällen die gleiche bleiben wird. Und da Divisionen recht zeitaufwendig sind, sollten wir versuchen, unnötige Divisionen (und dazu gehört auch der Modulo) zu vermeiden.

Wenn wir den Bitmapparameter komplett streichen und der `Map`-Klasse eine `setTargetBitmap()`-Methode spendieren, können wir dieses Problem umgehen.

```
void Map::setTargetBitmap(BITMAP *dest) {
    this->dest = dest;

    visibleTilesX = MIN(this->w, dest->w / tileWidth
                  + dest->w % tileWidth  == 0 ? 0 : 1);
    visibleTilesY = MIN(this->h, dest->h / tileHeight
```

```
                    + dest->h % tileHeight == 0 ? 0 : 1);
}

BITMAP *Map::getTargetBitmap() {
    return dest;
}

void Map::draw() {
    int pos = 0;

    for (int y=0; y < visibleTilesX; ++y) {
        for (int x=0; x < visibleTilesX; ++x) {
            drawTile(dest,
                     x*tileWidth, y*tileHeight,
                     data[pos]);
            ++pos;
        }
    }
}
```

Damit hätten wir den ersten Teil des Problems bereits gelöst. Jetzt geht es noch darum, den richtigen Teil der Karte anzuzeigen.

Da wir die Karte von links oben nach rechts unten anzeigen, brauchen wir nur zu wissen, mit welchem Feld wir anfangen müssen. Ab da ist dann alles wie gehabt.

```
void Map::draw(int wx, int wy) {
    int pos = 0;

    wx = MID(0, wx, maxX);
    wy = MID(0, wy, maxY);

    int ox = wx % tileWidth;
    int oy = wy % tileHeight;
    wx /= tileWidth;
    wy /= tileHeight;
    for (int y=0; y < visibleTilesY+1; ++y) {
        pos = wx + (y+wy) * w;
        for (int x=0; x < visibleTilesX+1; ++x) {
            drawTile(dest,
                     x*tileWidth-ox, y*tileHeight-oy,
                     data[pos]);
            ++pos;
        }
    }
}
```

Zu Beginn der abgewandelten `Map::draw()`-Methode stellen wir zuerst sicher, dass der übergebene Parameter gültig ist, dann wandeln wir die globalen Koordinaten in Feldkoordinaten um, indem wir durch die Breite bzw. Höhe einer Kachel teilen.

Die Position innerhalb des Daten-Arrays wird nun für jede Zeile neu berechnet, da jetzt nur noch die Spalten einer Zeile direkt nebeneinander im Speicher liegen.

Auch zeichnen wir nun eine Kachel mehr in jeder Richtung, da wir zwar maximal `visibleTiles` vollständige Kacheln sehen können, aber immer ein Teil einer weiteren Kachel teilweise zu sehen ist.

Auch die `Sprite::draw()`-Methode muss leicht abgewandelt werden, damit wir die globale Position des Sprites in eine Position auf dem Bildschirm umwandeln können.

```
void Sprite::draw(BITMAP *dest, int ox, int oy) {
    if (!frames) {
        return;
    }
    masked_blit(frames, dest,
                curFrame * frameWidth,
                curDirection * frameHeight,
                x-ox, y-oy,
                frameWidth, frameHeight);
}
```

Damit all diese Änderungen auch zusammenarbeiten, ist noch eine kleine Modifikation am Hauptprogramm nötig:

```
if (needsRedraw) {
    int x = MID(0,
                (hero->getX() - SCREEN_W/2),
                MAP_W * TILE_W - SCREEN_W);
    int y = MID(0,
                (hero->getY() - SCREEN_H/2),
                MAP_H * TILE_H - SCREEN_H);
    map.draw(x, y);
    hero->draw(doubleBuffer, x,y);

    needsRedraw = false;
    show();
}
```

Der letzte Puzzlestein ist eine größere Karte, denn sonst können wir das Scrolling ja gar nicht testen.

```
const int MAP_W  = 40;
const int MAP_H  = 17;
const int TILE_W = 32;
const int TILE_H = 32;


char tileMapStr[MAP_W * MAP_H+1] =
//0...4....90...4....90...4....90...4....9
 "1--------------2   1--------------2   " // 0
 "!..............!   !..............!   " // 1
 "!..............!   !..............!   " // 2
 "!..............3---4..............3--2" // 3
 "!....................................!" // 4
 "3----2...............................!" // 5
 "      !..............................!" // 6
 "      !..............................!" // 7
 "      !..............................!" // 8
 "      !..............................!" // 9
 "  1--4...............................!" // 10
 "  !.................................!" // 11
 "  !.................................!" // 12
 "  3-----2............................!" // 13
 "        !...................1---------4" // 14
 "        !...................!          " // 15
 "        3--------------------4          " // 16
 ;
```

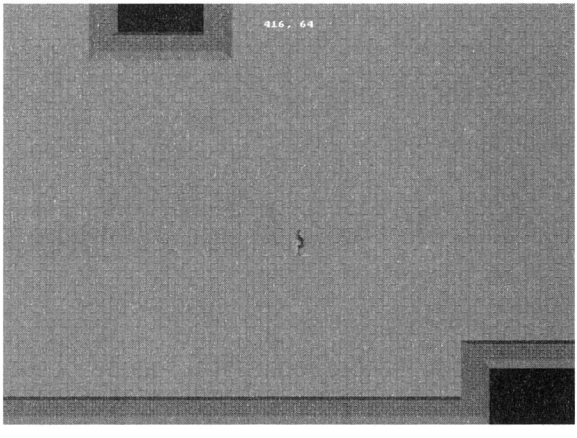


Abbildung 25.3: Die scrollende Karte

Das vollständige Testprogramm finden Sie natürlich wie immer auf der CD.

Editoren für Karten

Nun ist das Erstellen von Karten mit ASCII-Zeichen sicherlich eine Möglichkeit schnell eine Map zu bauen, allerdings gibt es auch einige Nachteile:

- ✔ Das Erstellen ist unkomfortabel.
- ✔ Sinnvolle Zeichen sind nur begrenzt vorhanden.
- ✔ Speicher wird doppelt belegt, einmal für den String, einmal für die Karte.
- ✔ Dynamisches Nachladen von Karten ist nicht direkt möglich.
- ✔ Zusatzinformationen können nur schwer untergebracht werden.
- ✔ Man hat kein direktes visuelles Feedback vom Aussehen der Karte.

Es wäre deutlich praktischer, wenn man seine Karten in einem Tool erstellen könnte, in dem man direkt sieht, wie die resultierende Map aussieht. Ein solches Tool zu entwickeln würde allerdings den Rahmen dieses Buches sprengen – und ist auch nicht notwendig, da es bereits einige Mapeditoren gibt, auf die man zurückgreifen kann. Eines dieser Tools ist Mappy.

Eine Version von Mappy finden Sie auf der CD zum Buch.

Mappy

Starten Sie Mappy und wählen Sie *File/ New Map...* im Menü aus. In der sich nun öffnenden Dialogbox wählen Sie *Nein*, damit Sie in das einfachere Dialogfeld zum Erstellen einer Karte kommen. Das Ergebnis sollte in etwa wie in Abbildung 25.4 aussehen.

Die Größe der Kacheln ist per Voreinstellung 32x32 Pixel. Diesen Wert können Sie direkt übernehmen. Die Kartengröße jedoch ist mit 100x100 recht großzügig. Bei einer Auflösung von 640x480 würde die daraus resultierende Karte über 33 Bildschirme füllen.

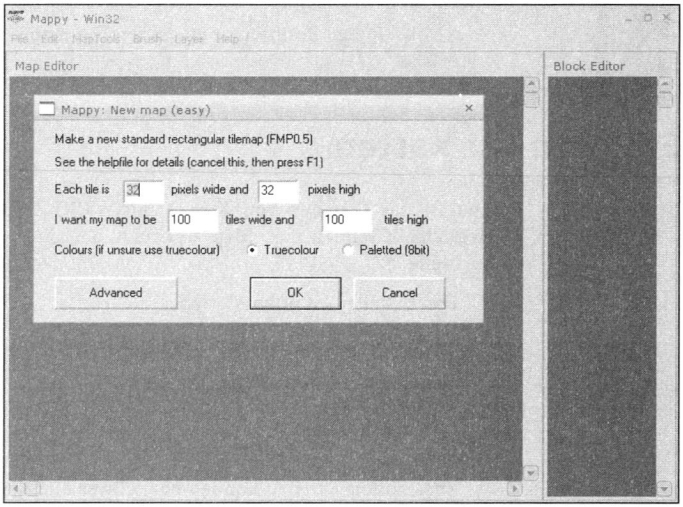


Abbildung 25.4: Neue Karte mit Mappy

Ändern Sie also die Kartengröße auf eine Breite von 60 Kacheln und eine Höhe von 45 Kacheln um. Bestätigen Sie mit *OK*. In der sich nun öffnenden Dialogbox bestätigen Sie nochmals mit *OK*.

Nun haben Sie eine leere Karte auf der linken Seite und einen fast leeren Bereich mit Kacheln auf der rechten Seite (siehe Abbildung 25.5).

Wählen Sie *File / Import*, um ein paar Kacheln zu laden. Wählen Sie die Bitmapdatei mit den Kacheln für das Verlies aus und bestätigen Sie. In der sich nun öffnenden Dialogbox bestätigen Sie ein weiteres Mal und werden mit einer Liste der Kacheln belohnt.

Sie können nun durch einen Klick eine Kachel auswählen und dann mit dieser auf der Karte malen.

Mappy erlaubt es Ihnen, Kacheln auf mehrere Ebenen zu setzen. Sie können im *Layer*-Menü neue Ebenen hinzufügen und dann auf dieser Ebene arbeiten. Wir werden in späteren Kapiteln diese Layer recht intensiv nutzen, es macht also durchaus Sinn, wenn Sie sich jetzt schon etwas mit den Layern auseinandersetzen (siehe Abbildung 25.6).

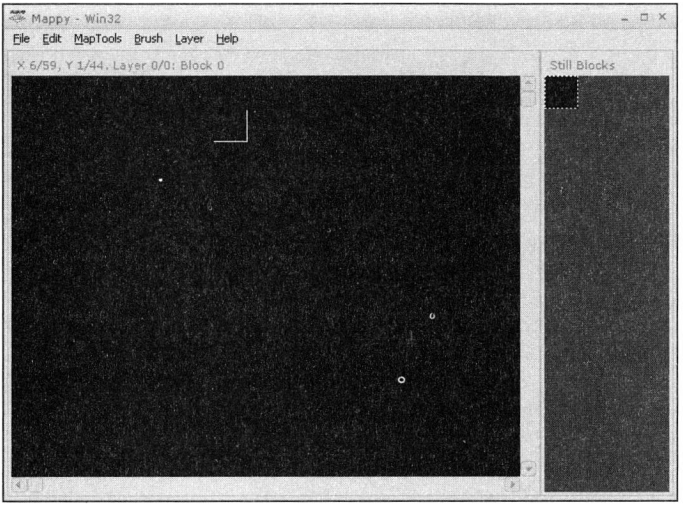


Abbildung 25.5: Eine neue Karte in Mappy

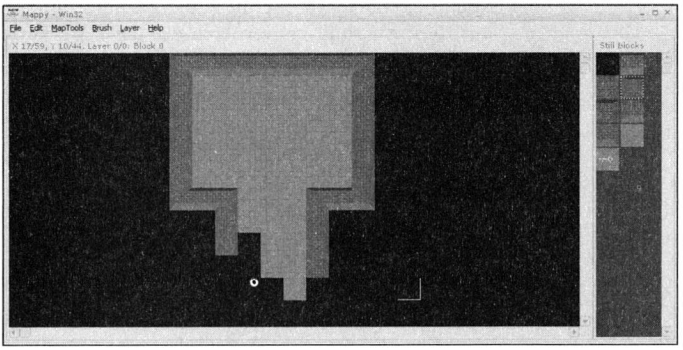


Abbildung 25.6: Erstellen der Karte

Bauen Sie sich eine Karte und speichern Sie sie dann mittels *File / Save* ab.

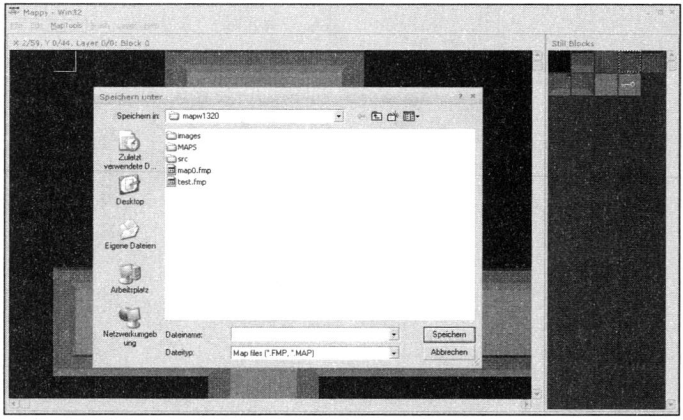


Abbildung 25.7: Speichern der Karte

Nun müssen wir unser Programm so erweitern, dass es Karten von der Festplatte lesen kann. Es gibt ein kleines Allegro Programm, das Mappys FMP-Dateien lesen kann.

Allerdings brauchen wir nicht die gesamte Funktionalität von Mappy. Aus diesem Grund konvertieren wir das FMP-Format in eine simplere Variante.

Wir sind derzeit ja nur an den reinen Kachelinformationen interessiert. Aus diesem Grund konvertieren wir das komplexe FMP-Format in ein sehr simples Dateiformat mit folgendem Aufbau:

```
int width;
int height;
int depth;
für alle y von 0 bis height-1
    für alle x von 0 bis width   1
         für alle z von 0 bis depth-1
            tile für Feld (x,y,z)
```

Wir werden dieses Format im Laufe der folgenden Kapitel etwas erweitern, aber dies ist die Startvariante.

Wie Sie sehen unterstützen wir bereits jetzt mehrere Ebenen – zumindest im Dateiformat. Die `Map`-Klasse müssen wir erst noch entsprechend an-

passen. Aber bevor wir das in Angriff nehmen können, müssen wir erst noch das Konverterprogramm erstellen.

Ich habe die Hilfsroutinen, die man von Mappys Webseite (*http://www.tilemap.co.uk/*) herunterladen kann leicht angepasst, damit sie auch funktionieren, ohne einen Grafikmodus zu setzen.

Stellen Sie aus diesem Grund sicher, dass Sie entweder die auf der CD enthaltenen Quellen nutzen, oder, wenn Sie neuen Code von Mappys Webseite heruntergeladen haben, Sie die Änderungen am Quellcode nachziehen.

Da ein Konverter allerdings nicht sehr interessant ist, kommt hier der Quellcode ohne weitere Umschweife:

```
#include <allegro.h>
#include <stdio.h>
#include <string.h>
#include "mappyal.h"

void usage() {
    printf("fmp2map - Converts a Mappy (*.fmp) map to a simple map
    format (*.map).\n");
    printf("\nThe resulting map will have the format:\n");
    printf("\tint32 width;\n");
    printf("\tint32 height;\n");
    printf("\tint32 depth;\n");
    printf("\twidth*height times: [depth * int32];\n\n");
    printf("Usage:\n\tfmp2map in.fmp [out.map]\n");
}

int main(int argc, char ** argv) {
    int x, y, z;
    int width, height;
    int depth = 0;
    long tile  = 0;
    int search = 0;
    int error = 0;
    char *filename;
    PACKFILE *f;

    if (argc != 2 && argc != 3) {
        usage();
        exit(1);
    }
```

```
allegro_init();
set_color_depth(16);
error = MapLoad(argv[1]);
if (error != 0) {
    printf("Unable to load mappy map: %s (%i)\n", argv[1],
    maperror);
    exit(1);
}

if (argc ==2) {
    filename = strdup(argv[1]);
    replace_extension(filename, argv[1], "map",
                      strlen(argv[1])+1);
}

/* Check the number of layers */
while (MapChangeLayer(depth) >= 0) {
    ++depth;
}
width  = mapwidth;
height = mapheight;
printf("Output to %s, size: %ix%ix%i\n",
       filename, width, height, depth);

// Kompression einschalten
f = pack_fopen(filename, "wp");

pack_fwrite(&width,  sizeof(int), f);
pack_fwrite(&height, sizeof(int), f);
pack_fwrite(&depth,  sizeof(int), f);

for (y=0; y < mapheight; y++) {
    for (x=0; x < mapwidth; x++) {
        for (z=0; z < depth; z++) {
            MapChangeLayer(z);
            tile = MapGetBlock(x,y)->bgoff;

            if (y==0) {
                for (search=0;
                     search < mapnumblockgfx;
                     search++) {
                    if ((long int) abmTiles[search]
                        == tile) {
                        pack_fwrite(&search,
                                    sizeof(int), f);
```

```
                        }
                    }
                }
            }

        }
    }

    pack_fclose(f);

    if (argc == 2) {
        free(filename);
    }

    MapFreeMem();

    return 0;
} END_OF_MAIN()
```

Der Konverter bietet zwar nicht sehr viel Luxus, hat aber immerhin eine kleine Hilfefunktion eingebaut. Wir benutzen diesmal Allegros eingebaute Kompression, um die Größe der erzeugten Map zu reduzieren.

Durch dieses kleine Tool können wir nun auf recht einfache Weise `FMP` Dateien in ein einfaches Format umwandeln. Und wenn wir `Makefiles` benutzen, dann können wir einen großen Teil der Arbeit auch automatisieren.

Wenn wir `make` mitteilen, dass `.map`-Dateien durch einen Aufruf von `fmp2map` aus `.fmp`-Dateien erstellt werden können und dann die `.map`-Dateien als Abhängigkeiten vom eigentlichen Spiel kennzeichnen, dann wird bei jedem Aufruf von `make` sicher gestellt, dass alle Karten auf dem aktuellen Stand sind.

Und genau für diesen Zweck hat `make` *Suffix-Regeln*. Die Suffix-Regel besteht aus der Ausgangsendung, gefolgt von der Zielendung und dem Aufruf des entsprechenden Tools. In unserem Fall also:

```
.fmp.map:
    fmp2map $<
```

Durch diese Suffix-Regel wird automatisch `fmp2map` aufgerufen, wenn eine Änderung an einer `fmp`-Datei festgestellt wird.

Die Karte laden

Zum Laden der Karte können wir beinahe den Code zum Abspeichern benutzen – wir müssen lediglich `pack_fwrite()` durch `pack_fread()` ersetzen.

```
int Map::load(const char* filename) {

    PACKFILE *f = pack_fopen(filename, "rp");
    if (!f) {
        return 0;
    }

    int width, height, depth;
    int tile;

    pack_fread(&width,  sizeof(int), f);
    pack_fread(&height, sizeof(int), f);
    pack_fread(&depth,  sizeof(int), f);

    create(width, height, depth);

    for (int y=0; y < height; ++y) {
        for (int x=0; x < width; ++x) {
            for (int z=0; z < depth; ++z) {
                pack_fread(&tile,  sizeof(int), f);
                setTileAt(x, y, z, tile);
            }
        }
    }

    pack_fclose(f);
    return 1;
}
```

Allerdings muss nun auch die `Map`-Klasse selber auf das Verwalten mehrerer Ebenen vorbereitet werden.

Die `create()`-Methode bekommt einen zusätzlichen Parameter (die Anzahl der Ebenen).

```
void Map::create(int w, int h, int d) {
    this->w = w;
    this->h = h;
    this->d = d;
    this->lineSize = w*d;
```

```
    this->size      = w*h*d;
    this->data      = new char[size];
    this->dest      = NULL;
    this->visibleTilesX = 0;
    this->visibleTilesY = 0;
}
```

Die `lineSize` speichert die Größe einer einzelnen Zeile und wird verwendet, um den Zugriff auf die Daten etwas zu beschleunigen.

Natürlich brauchen wir auch Funktionen, um auf die Tiles zugreifen zu können.

```
int  Map::getTileAt(int tileX, int tileY, int tileZ) {
    int pos = tileZ + tileX * d+ tileY * lineSize;
    if (pos >=0 && pos < size) {
        return data[pos];
    }
    return 0;
}

void Map::setTileAt(int tileX, int tileY, int tile) {
    int pos = tileX * d + tileY * lineSize;
    if (pos >=0 && pos < size) {
        data[pos] = (char) (tile & 0xff);
    }
}

void Map::setTileAt(int tileX, int tileY, int tileZ, int tile) {
    int pos = tileZ + tileX * d + tileY * lineSize;
    if (pos >=0 && pos < size) {
        data[pos] = (char) (tile & 0xff);
    }
}
```

Die Reihenfolge, in der die Daten gespeichert werden, ist anfangs etwas ungewohnt. Allerdings führt sie dazu, dass die Tiles, die kurz nacheinander gezeichnet werden, auch dicht im Speicher beisammen liegen. Dadurch kann der Speicherzugriff effizienter erfolgen, und wir gewinnen etwas Zeit.

```
void Map::draw(int wx, int wy) {
    int pos = 0;

    wx = MID(0, wx, maxX);
    wy = MID(0, wy, maxY);
```

```
    int ox = wx % tileWidth;
    int oy = wy % tileHeight;
    wx /= tileWidth;
    wy /= tileHeight;

    for (int y=0; y < visibleTilesY+1; ++y) {
        pos = wx * d + (y+wy) * lineSize;
        for (int x=0; x < visibleTilesX+1; ++x) {
            for (int z=0; z < d; ++z) {
                drawTile(dest, x*tileWidth-ox,
                            y*tileHeight-oy, data[pos]);
                ++pos;
            }
        }
    }
}
```

Jetzt können Sie Karten mit Mappy erstellen und in Ihnen herum laufen. Sie können sogar mehrere Ebenen benutzen. Warum das eine gute Sache ist, erfahren Sie im nächsten Abschnitt.

Mehrere Ebenen

Greg Taylor hat im Jahre 1995 sein »Tile Based Games FAQ" (zu deutsch; »Häufig gestellten Fragen zum Thema auf Kacheln basierende Spiele«) im Usenet veröffentlicht. Das *TileFAQ*, wie es kurz genannt wird, ist bis heute eines der grundlegenden Informationsquellen für angehende Spieleentwickler.

Taylor geht von drei Ebenen, die vor den Sprites gezeichnet werden, aus, und schlägt eine weitere vor, die nach den Sprites gezeichnet werden soll, um damit zum Beispiel Dächer zu simulieren.

Ebenen Name	Nutzung für
Grund	Grass, Dreck, Steine, Wasser
Dekoration	Dreckspritzer, Übergänge, Steine
Objekte	Schwerter, Truhen, Schlüssel

Tabelle 25.1: Ebenenarten

Die Dekorationsebene wird verwendet, um die Grundtiles abwechslungsreicher zu gestalten. So könnten ein paar Steine auf der Dekorationsebene die regelmäßige Struktur der Grundebene aufreißen und für Abwechslung sorgen.

Wenn Sie einen Übergang von Erde zu Fels brauchen, dann platzieren Sie die Erd- und Felstiles auf der Grundebene und machen den Übergang durch eine Tile auf der Dekorationsebene fließender.

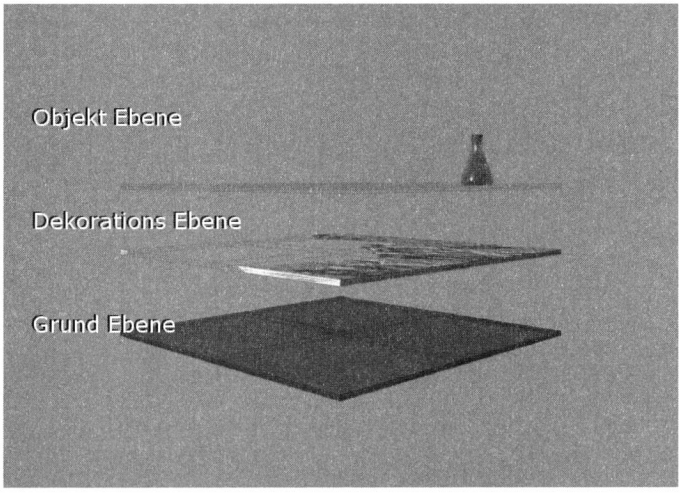


Abbildung 25.8: Ebenen im Überblick

Wie viele Ebenen Sie benutzen, ist natürlich Ihre Entscheidung. Einige meinen, dass zwei Ebenen für beinahe alle Fälle ausreichen, andere sind der Meinung, dass es mindestens fünf Ebenen sein sollten.

Lassen Sie sich davon nicht verwirren. Benutzen Sie so viele Ebenen wie Sie wollen. Sie brauchen nur eine? Dann nehmen Sie nur eine. Brauchen Sie 10? Auch kein Problem, dann nehmen Sie zehn.

Machen Sie sich keine zu großen Gedanken darüber. Ebenen sind wie Tiles selbst nur ein Mittel zum Zweck. Sie sollen Ihnen helfen und Sie nicht einengen.

Für das letzte Beispiel in diesem Kapitel verwende ich zwei Ebenen. Eine Ebene für den Hintergrund und eine weitere Ebene für die Objekte.

Mappy für Karten mit mehreren Ebenen

Wenn wir eine weitere Ebene hinzufügen wollen, dann können wir das über das Menü *Layer/ Add Layer* erreichen. In diesem Menü können Sie auch die derzeit aktive Ebene auswählen.

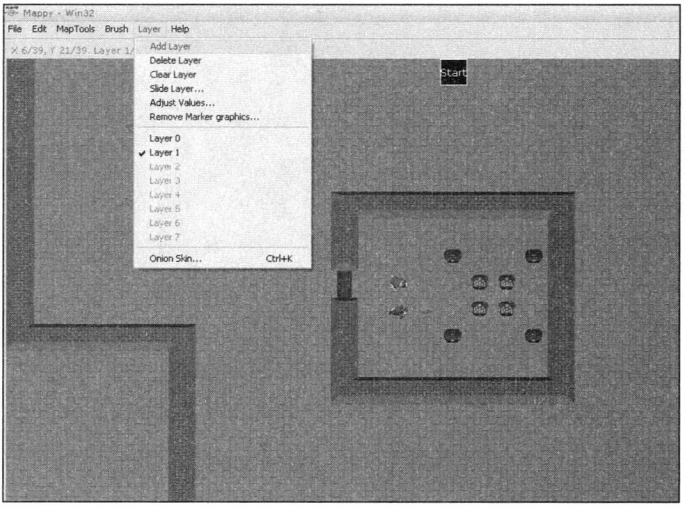


Abbildung 25.9: Das Layermenü von Mappy

Ein weiterer, sehr wichtiger Punkt ist *Layer/ Onion Skin.* Sie können den entsprechenden Dialog auch mittels Strg + K aufrufen.

Sollte die Karte schwarz werden, sobald Sie eine Ebene hinzufügen, dann ist dies ein eindeutiges Zeichen dafür, dass *Onion Skinning* nicht aktiviert ist. Rufen Sie in diesem Fall den *Onion-Skinning*-Dialog auf, und stellen Sie sicher, dass die Checkbox *Enable Onion Skin* auch ein Häkchen hat.

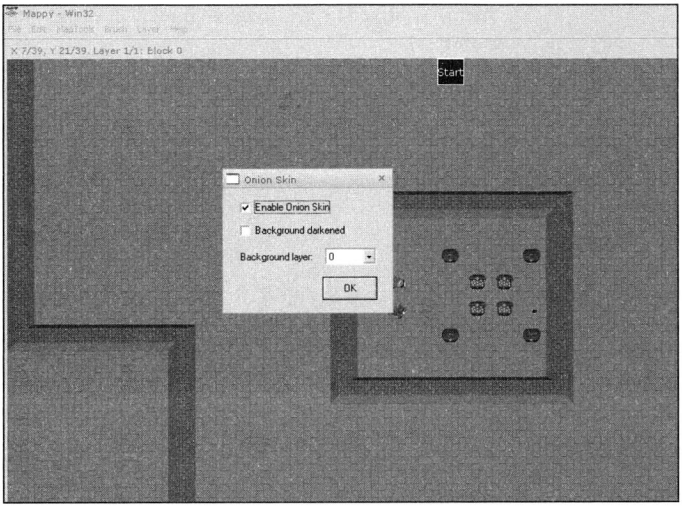


Abbildung 25.10: Onion Skinning

Damit Sie die Vorteile von Ebenen auch so richtig ausnützen können, brauchen wir auch ein Tileset, das einige Objekte enthält. Der talentierte Pixel-Künstler Zooey »Zoggles« Ball hat uns einige Grafiken zur Verfügung gestellt.

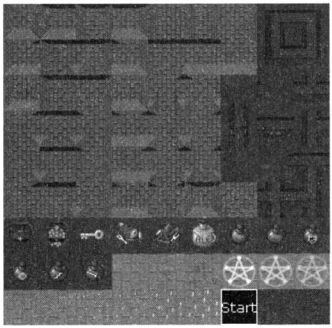


Abbildung 25.11: Dungeon Tiles

Mehr Grafiken von Zoggles finden Sie auf *http://zoggles.dyndns.org/zoggles/*.

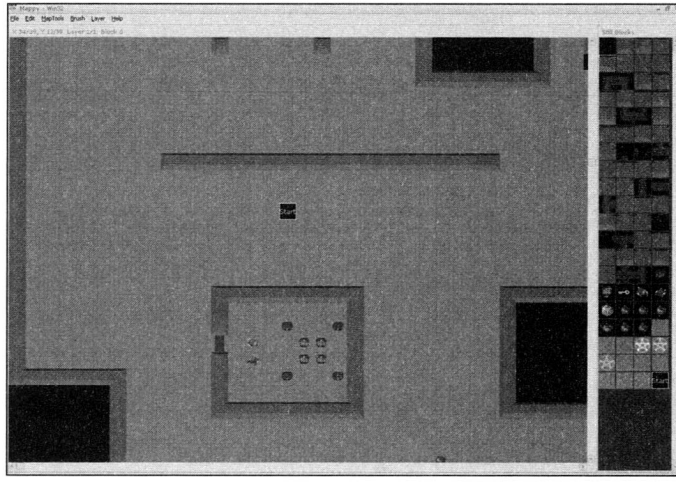


Abbildung 25.12: Die neuen Grafiken in Aktion

Abgesehen von den Objekten und Wänden gibt es noch ein weiteres Tile, das erklärungsbedürftig ist: das Start Tile.

Diese besondere Kachel markiert die Startposition des Helden in diesem Level. Beim Laden des Levels überprüfen wir die zweite Ebene, und wenn wir das Start Tile finden, ersetzen wir es in der Karte durch ein leeres Tile, aber merken uns die Position.

Das Problem ist, dass beim Laden der Karte noch nicht unbedingt auch schon die korrekten Kacheln gesetzt sind. Allerdings können wir dieses Problem recht einfach lösen. Wenn das Start Tile immer das Tile mit dem höchsten Tile Index ist (also die letzte Kachel in dem Tile Grafik File), und jede Karte eine Start Tile hat, dann können wir davon ausgehen, dass das Tile mit dem höchsten Index in der Karte die Startposition ist.

```
int Map::load(const char* filename) {

    PACKFILE *f = pack_fopen(filename, "rp");
```

```
    if (!f) {
        return 0;
    }

    int width, height, depth;
    int tile;

    pack_fread(&width,  sizeof(int), f);
    pack_fread(&height, sizeof(int), f);
    pack_fread(&depth,  sizeof(int), f);

    create(width, height, depth);

    int maxTile = -1;
    int maxX = 0;
    int maxY = 0;

    for (int y=0; y < height; ++y) {
        for (int x=0; x < width; ++x) {
            for (int z=0; z < depth; ++z) {
                pack_fread(&tile,  sizeof(int), f);
                setTileAt(x, y, z, tile);
                if (z == depth-1 && tile > maxTile) {
                    maxTile = tile;
                    maxX = x;
                    maxY = y;
                }
            }
        }
    }
    if (maxTile > 0) {
        // Löschen des Start tiles
        setTileAt(maxX, maxY, depth-1, 0);
        startX = maxX;
        startY = maxY;
    }

    pack_fclose(f);

    return 1;
}
```

Nach dem Laden der Karte können wir also die Startposition des Helden direkt aus der Karte lesen. Praktische Sache.

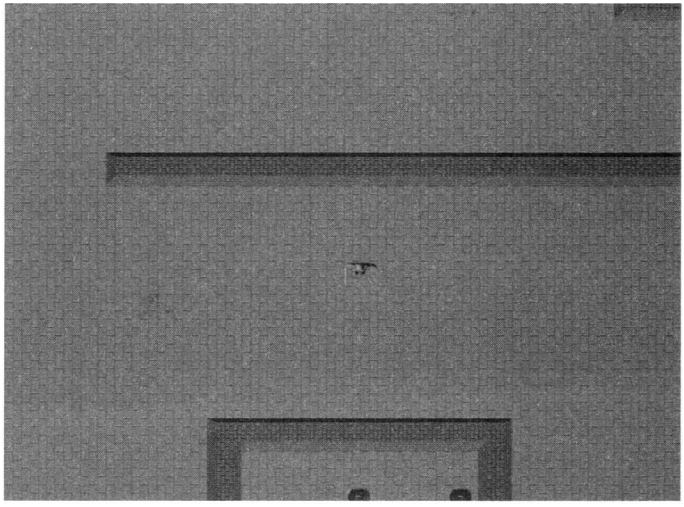


Abbildung 25.13: Der Held an der im Editor festgelegten Startposition

```
Map map;
map.load("level0.map");
map.setTiles(tiles, TILE_W, TILE_H);
map.setTargetBitmap(doubleBuffer);

hero->setMap(&map);
hero->setPosition(map.getStartX() * TILE_W,
                  map.getStartY() * TILE_H);
```

Derzeit ist die Begehbarkeit noch hart in der `Map`-Klasse codiert. Da wir nun andere Tiles haben, ist es notwendig, auch die `Map`-Klasse anzupassen.

```
bool Map::isWalkable(int tileX, int tileY) {
    int tile = getTileAt(tileX, tileY);
    return (tile >=54);
}
```

Wenn Sie sich jetzt sagen: »Wie? Ich muss den Quellcode ändern, sobald ich neue Tiles hinzufüge? Das ist aber nicht so toll...«, dann haben Sie 100% Recht. Dies ist in der Tat keine gute Sache, und wir sollten das möglichst schnell ändern.

Bei dieser Gelegenheit verknüpfen wir auch gleich den Level mit der benötigten Karte und den benötigten Tiles.

Legen Sie eine Datei `game.ini` an und füllen Sie sie mit folgenden Zeilen:

```
[level0]
map=level0.map
tiles=tile.tga
walkable=54
```

Level 0 nutzt also die Karte `level0.map` und die in `tile.tga` gespeicherten Kacheln. Alle Tiles ab Index 54 sind begehbar. Wenn wir diese Information nun noch im Programm einlesen, dann sind wir einer flexiblen Engine einen großen Schritt näher gekommen.

```
    push_config_state();
    set_config_file("game.ini");

    BITMAP *tiles   = load_bitmap(
                        get_config_string( "level0",
                                           "tiles",
                                           "tile.tga"),
                      NULL);
    map.load(get_config_string("level0",
                               "map",
                               "level0.map"));
    map.setTiles(tiles, TILE_W, TILE_H,
                 get_config_int("level0",
                                "walkable",
                                0));
    map.setTargetBitmap(doubleBuffer);
    pop_config_state();
```

Wie Sie sehen, haben wir `Map::setTiles()` um einen Parameter erweitert.

```
void Map::setTiles(BITMAP *tiles, int tw, int th, int firstWalkable)
{
    this->tiles = tiles;
    tileWidth   = tw;
    tileHeight  = th;

    this->firstWalkable = firstWalkable;
}
```

Wir setzen in der Routine eine Variable auf den Index des ersten Tiles, das begehbar ist. Nun kann die `isWalkable()` Methode auf die bisher hart codierte Konstante verzichten.

```
bool Map::isWalkable(int tileX, int tileY) {
    int tile = getTileAt(tileX, tileY);
    return (tile >=firstWalkable);
}
```

Der Held erscheint an der korrekten Stelle der Karte, wir können durch große Gebiete streifen, aber bis jetzt sind die in der Karte platzierten Objekte nur schmückendes Beiwerk. Darum kümmern wir uns im nächsten Kapitel.

Den Quellcode zu diesem Kapitel finden Sie auf der beiliegenden CD.

26 Objekte

Es wird Zeit, die Kerker zum Leben zu erwecken! In diesem Kapitel lernen Sie, wie Ihr Held mit der Umwelt interagieren kann. Am Ende dieses Kapitels werden Sie in der Lage sein, beliebige Arten von Objekten in Ihrem Kerker zu platzieren. Auch Türen und verschlossene Schatztruhen sind dann kein Problem mehr.

Gegenstände aufnehmen

Ein »Gegenstand« ist bisher eine Kachel auf der Objektebene der Karte. Oder anders gesagt: Wenn das Tile in der Objektebene an der Position des Spielers ungleich 0 ist, dann steht er mitten in einem Objekt.

Der erste Schritt besteht darin, das Objekt von der Karte verschwinden zu lassen und dem Spieler in irgendeiner Weise mitzuteilen, dass er das Objekt aufgenommen hat. Am häufigsten wird hierzu ein Soundeffekt eingesetzt, manchmal unterstützt durch einen kleinen grafischen Effekt.

Das Objekt »verschwinden« zu lassen, ist recht einfach. Sobald wir die entsprechende Kachel auf 0 gesetzt haben, ist es weg.

```
// In Sprite::update()
if (!map->isWalkable(tx, ty)) {
    x = ox;
    y = oy;
} else {
    tx = (x+ frameWidth  /2)/tw;
    ty = (y+ frameHeight /2)/th;
    int layer = map->getObjectLayer();
    int obj = map->getTileAt(tx, ty, layer);
    if (obj) {
        map->setTileAt(tx, ty, layer, 0);
    }
}
```

Da das `Sprite` nicht wissen kann, auf welcher Ebene eine Karte die Objekte speichert, fragt es vorher erst einmal nach, überprüft dann den Tile Index und löscht gegebenenfalls das Objekt.

Das objectLayer ist ein einfaches Attribut der Karte. Es kann von außen gesetzt und abgefragt werden.

```
void Map::setObjectLayer(int objectLayer) {
    this->objectLayer = objectLayer;
}

int Map::getObjectLayer() {
    return objectLayer;
}
```

Diese Änderungen reichen, um ein Sprite Objekte aufsammeln zu lassen. Aber wie speichern wir das Objekt? Und wie informieren wir den Spieler, dass wir ein Objekt aufgenommen haben? Wie wird festgelegt, welches Tile welches Objekt repräsentiert?

Vom Sprite zum Charakter

Wir sind an einem Punkt angelangt, an dem es Zeit wird Spezialisierungen vorzunehmen. Spezialisierungen sind in beinahe allen Fällen abgeleitete Klassen. Die Sprite-Klasse wird die Basisklasse für alle Spezialisierungen eines Sprites. Die erste Klasse, die wir von Sprite ableiten, ist PlayerCharacter.

Um sicher zustellen, dass die Polymorphie auch wie gewünscht funktioniert, werden alle Methoden von Sprite als virtual gekennzeichnet.

Es ist nicht unbedingt nötig, alle Funktionen als virtual zu kennzeichnen. Allerdings sollten virtuelle Funktionen immer der Ausgangspunkt sein. Sie können die Klassen am Ende eines Projektes immer noch optimieren und dann nur die benötigten Funktionen als virtuell kennzeichnen.

Wir erweitern diese Schnittstelle um zwei weitere Methoden. Die Methode incFrame() wird von update() aufgerufen, sobald sich der aktuelle Frame ändert. Abgeleitete Klassen können hier dann Sounds abspielen oder auf andere Weise reagieren.

Wenn ein Objekt aufgenommen wird, dann soll objectTaken() aufgerufen werden, damit die abgeleitete Klasse entsprechend reagieren kann (Sounds abspielen, Punktestand erhöhen etc.).

Damit die abgeleiteten Klassen Zugriff auf die Attribute von Sprite haben, werden diese als `protected` deklariert. Die spielerspezifischen Variablen wie der `stepSnd` und die entsprechende `setStepSnd()`-Methode wurden entfernt.

```
#ifndef SPRITE_HEADER
#define SPRITE_HEADER

#include <allegro.h>
#include "util.h"
#include "map.h"

class Sprite {
protected:
    int frameWidth;
    int frameHeight;
    int curFrame;
    int curDirection;
    bool moving;

    BITMAP *frames;
    int countFrames;

    int *deltaX;
    int *deltaY;

    int x;
    int y;

    float speed;
    float updateCounter;


    Map *map;
public:
    enum {
        DOWN  = 0,
        UP    = 1,
        RIGHT = 2,
        LEFT  = 3
    };
    Sprite() : frameWidth(0), frameHeight(0),
               curFrame(0)  , curDirection(0),
               frames(NULL) , countFrames(0),
```

```
                deltaX(NULL) , deltaY(NULL),
                x(0)         , y(0),
                speed(0.1)   , updateCounter(0.0),
                map(NULL)        {
    }

    Sprite(BITMAP *img, int fw, int fh);

    virtual ~Sprite();

    virtual void setMoving(bool move);
    virtual bool isMoving();
    virtual void setDirection(int dir);

    virtual void setX(int x);
    virtual void setY(int y);
    virtual void setPosition(int x, int y);

    virtual int  getX();
    virtual int  getY();

    virtual void setDeltaX(int frame, int delta);
    virtual void setDeltaY(int frame, int delta);

    virtual void setSpeed(float speed);

    virtual void setMap(Map* map);
    virtual Map *getMap();

    virtual void update();
    virtual void draw(BITMAP *dest, int ox, int oy);

protected:
    virtual void objectTaken();
    virtual void incFrame(int delta);
};

#endif
```

Die Änderungen am Quellcode fallen recht gering aus. In der `update()`-Methode wird nun `incFrame()` aufgerufen, anstatt die `Frame`-Variable direkt zu erhöhen. Und nach dem Aufsammeln eines Objektes wird `objectTaken()` aufgerufen. Diese Methoden sind funktional identisch mit dem bisherigen Code:

```
void Sprite::incFrame(int delta) {
    curFrame+= delta;
    if (curFrame >= countFrames) {
        curFrame -= countFrames;
    } else if (curFrame < 0) {
        curFrame += countFrames;
    }
}

void Sprite::objectTaken() {
}
```

Aber jetzt können wir diese Methoden in der PlayerCharacter-Klasse überladen und alle zum Spieler gehörenden Sounds in dieser Klasse kapseln. Zu diesem Zweck nutzen wir ein Array, das alle SAMPLE Zeiger speichert. Für den Zugriff auf dieses Array nutzen wir ein anonymes enum als Ersatz für Klassenkonstanten.

```
#ifndef PLAYERCHARACTER_HEADER
#define PLAYERCHARACTER_HEADER

#include <allegro.h>
#include "util.h"
#include "map.h"
#include "sprite.h"

class PlayerCharacter : public Sprite {
    enum {
        FOOTSTEPS    = 0,
        YEAH         = 1,
        SAMPLE_COUNT = 2,
    };

    SAMPLE* samples[SAMPLE_COUNT];

public:

    PlayerCharacter();
    PlayerCharacter(BITMAP *img, int fw, int fh);
    ~PlayerCharacter();

    bool create();

protected:
    void objectTaken();
```

```
    void incFrame(int delta);
};

#endif
```

Nach dem Aufruf von `create()` stehen der Klasse alle Samples zur Verfügung. Ein Aufruf von `playSample(samples[FOOTSTEPS], 255));` spielt dann die Schrittgeräusche in voller Lautstärke ab.

```
#include "playerCharacter.h"

PlayerCharacter::PlayerCharacter() {
    create();
}

PlayerCharacter::PlayerCharacter(BITMAP *img, int fw, int fh) :
Sprite(img, fw, fh) {
    create();
}


PlayerCharacter::~PlayerCharacter() {
    for (int a=0; a < SAMPLE_COUNT; ++a) {
        if (samples[a]) {
            destroy_sample(samples[a]);
            samples[a] = NULL;
        }
    }
}

bool PlayerCharacter::create() {
    char* filenames[] = {
        "step.wav",
        "collect.wav",
    };
    for (int a=0; a < SAMPLE_COUNT; ++a) {
        samples[a] = load_sample(filenames[a]);
    }
    return true;
}

void PlayerCharacter::objectTaken() {
    playSound(samples[YEAH], 250);
}
```

```
void PlayerCharacter::incFrame(int delta) {
    // Aufruf der Basis Klasse
    Sprite::incFrame(delta);
    if ((curFrame & 1)==1) {
        playSound(samples[FOOTSTEPS], 180);
    }
}
```

Wenn Sie Methoden überladen, dann denken Sie daran, die Methode der Basisklasse aufzurufen (es sei denn, Sie wollen das Verhalten komplett ändern, anstatt es nur zu erweitern).

Im letzten Schritt gilt es, das Hauptprogramm so abzuwandeln, dass es anstatt der `Sprite`-Klasse die `PlayerCharacter`-Klasse nutzt. Dies kann durch das Ändern einer Zeile erreicht werden. Anstatt

```
Sprite *hero = new Sprite(heroSpr, TILE_W, TILE_H);
```

deklarieren und initialisieren wir `hero` nun wie folgt:

```
PlayerCharacter *hero = new PlayerCharacter(
                            heroSpr, TILE_W, TILE_H);
```

Nach dieser Änderung funktioniert alles wie gehabt ... mit einer Änderung: Beim Aufsammeln von Objekten gibt der Spielercharakter nun eine freudige Äußerung von sich.

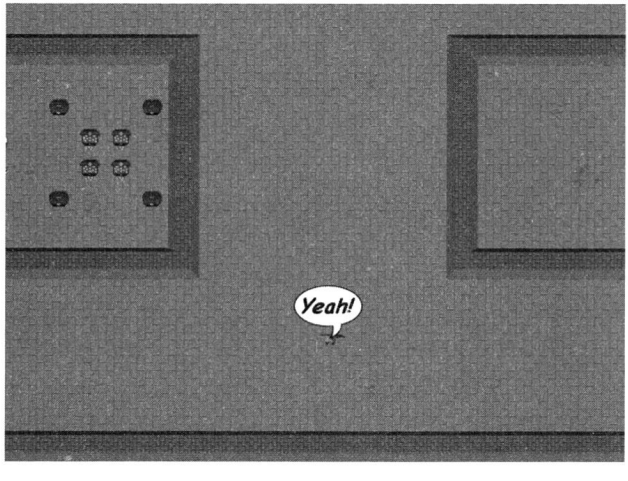


Abbildung 26.1: Ein zufriedener Barbar

Die Sprechblase in Abbildung 26.1 war anfangs eigentlich nur als Gag gedacht, und in einem Grafik Programm hinzugefügt. Allerdings fand ich das Ergebnis recht ansprechend – und warum sollte man also nicht ein paar kleine Sprechblasen hinzufügen?

Erweitern wir die `PlayerCharacter` Klasse entsprechend.

```
class PlayerCharacter : public Sprite {
    enum {
        FOOTSTEPS    = 0,
        YEAH         = 1,
        SAMPLE_COUNT = 2,
    };

    SAMPLE* samples[SAMPLE_COUNT];
    BITMAP* bubbles[SAMPLE_COUNT];

    int bubbleTimeOut;
    int curBubble;

public:

    PlayerCharacter();
    PlayerCharacter(BITMAP *img, int fw, int fh);
    ~PlayerCharacter();

    bool create();

    void update();
    void draw(BITMAP *dest, int ox, int oy);

protected:
    void objectTaken();
    void incFrame(int delta);
};
```

Wir benötigen einen Zähler, der angibt wie lange die Sprechblase noch zu sehen sein soll (`bubbleTimeOut`), den Index der aktuellen Sprechblase (`curBubble`) und ein Array von `BITMAP`-Zeigern, um die Grafiken zu verwalten. Damit die Sprechblase auch angezeigt wird, müssen wir `draw()` überladen und der Zähler muss in `update()` aktualisiert werden.

Schritt Nummer eins ist, die Grafiken zu laden:

```
bool PlayerCharacter::create() {
    char* filenames[] = {
        "step.wav",
```

```
        "collect.wav",
    };
    char* bubblefiles[] = {
        NULL,
        "yeah.tga",
    };

    for (int a=0; a < SAMPLE_COUNT; ++a) {
        samples[a] = load_sample(filenames[a]);
        if (bubblefiles[a] != NULL) {
            bubbles[a] = load_bitmap(bubblefiles[a],
                                     NULL);
        } else {
            bubbles[a] = NULL;
        }
    }
    bubbleTimeOut = 0;
    curBubble = -1;
    return true;
}
```

Die Sprechblase soll das Geräusch visualisieren. Allerdings sollte nicht jedes Geräusch auch eine Sprechblase haben (Bei jedem Schritt ein »Tapp!« wäre zwar anfangs recht lustig, stört aber nach recht kurzer Zeit.). Man sollte also hier variieren können. Aus diesem Grund signalisiert eine `NULL` im `bubblesFile`-Array, dass es für dieses Geräusch keine passende Sprechblase gibt.

Der `bubbleTimeOut` und die `curBubble` werden in `objectTaken()` gesetzt:

```
void PlayerCharacter::objectTaken() {
    playSound(samples[YEAH], 250);
    curBubble = YEAH;
    // Eine Sekunde lang anzeigen
    bubbleTimeOut = 60;
}
```

Die `update()`-Methode stellt sicher, dass die Sprechblase auch wieder verschwindet, und `draw()` zeigt sie an:

```
void PlayerCharacter::update() {
    Sprite::update();
    if (bubbleTimeOut) {
        --bubbleTimeOut;
        if (bubbleTimeOut == 0) {
            curBubble = -1;
```

```
        }
    }
}


void PlayerCharacter::draw(BITMAP *dest,
                           int ox, int oy) {
    Sprite::draw(dest, ox, oy);
    if (curBubble >= 0) {
        draw_sprite(dest, bubbles[curBubble],
                    x-ox + frameWidth/2
                         - bubbles[curBubble]->w/2,
                    y-oy + frameHeight/2
                         - bubbles[curBubble]->h);
    }
}
```

Der Mittelpunkt der unteren Kante der Sprechblase wird zu dem Zentrum des Sprites ausgerichtet.

Verschiedene Objekte

Eigentlich sollten die Objekte nicht nur von der Karte verschwinden, sondern eine bestimmte Wirkung erzielen. So könnten Tränke einen schneller oder stärker machen, Essen sollte die Gesundheit wiederherstellen und mit Schlüsseln sollte man Truhen und Türen öffnen können.

Wir brauchen also eine Zuordnung von logischen Objekten wie Schlüsseln und Tränken zu Tiles. Es bietet sich an, zu diesem Zweck die `game.ini` Datei zu erweitern. Wir können in ihr all die Informationen speichern, die bisher fest im Hauptprogramm verdrahtet waren – und dann die Tilezuordnungen hinzufügen.

```
[level0]
map=level0.map
tiles=tile.tga
objectLayer=1

[tile.tga]
tileWidth=32
tileHeight=32

walkable=54
```

```
lockedChest=55
openChest=56
key=57
food25=58
food50=59
food75=60
health=61
mana=62
shield=63
attack=64
speed=65
damage=66
door=34 35

defaultScore=100
lockedChestScore=500
shieldScore=250
attackScore=400
speedScore=300
damageScore=600
```

Für jeden Level (in diesem Fall gibt es nur einen Level) wird die Karte, die Tiles und die Objektebene gespeichert. Der Name der Tilegrafikdatei ist gleichzeitig auch der Schlüssel für die Kachelinformationen. Dazu gehören die Höhe und Breite der Kacheln, das erste betretbare Tile und die Objektzuordnungen.

Falls mehrere Tiles zum gleichen Objekttyp gehören, dann können Sie diese in einer Reihe hinter dem Objekttypnamen aufzählen.

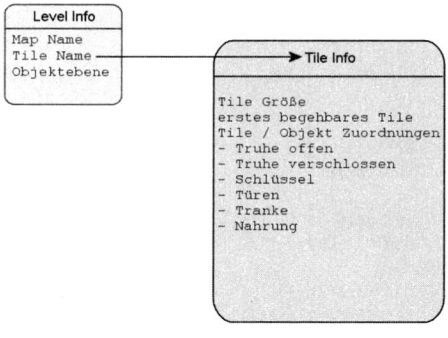


Abbildung 26.2: Aufbau der Game.ini Datei

Die »Spielwelt« besteht aus den Leveldaten, der Karte und allen Metainformationen zu diesen beiden Dateien. Die Metainformationen stehen in der `game.ini`-Datei und sollten von der Klasse, welche die Spielwelt verwaltet, aus dieser ausgelesen werden.

Die `World`-Klasse stellt das Bindeglied zwischen der Karte und den Tiles dar.

```
#ifndef WORLD_HEADER
#define WORLD_HEADER

#include <allegro.h>
#include "map.h"

class World {

public:
    enum {
        NONE            = 0,
        KEY             = 1,
        LOCKED_CHEST    = 2,
        OPEN_CHEST      = 3,
        FOOD25          = 4,
        FOOD50          = 5,
        FOOD75          = 6,
        HEALTH_POTION   = 7,
        MANA_POTION     = 8,
        SHIELD_POTION   = 9,
        SPEED_POTION    =10,
        STRENGTH_POTION =11,
        DOOR            =12,
    };
    enum {
        OBJECT_COUNT = 12
    };

    World();
    ~World();

    int getObjectID(int tile);
    bool readLevel(int id);

    Map* getMap();
    BITMAP *getTiles();
    int getFirstWalkable();
```

```
private:
    int  tileCount;
    int* tile2object;
    BITMAP *tiles;
    Map    *map;

    int tileWidth;
    int tileHeight;
    int firstWalkable;

};

#endif
```

Die beiden interessanten Methoden hier sind die `int World::getObjectID(int tile)`-Methode, welche die Objekt ID eines bestimmten Tiles zurückgibt, und die `bool World::readLevel(int id)`-Methode, welche die Informationen aus der `game.ini`-Datei ausliest.

Schauen wir uns zuerst die kleinere der beiden Methoden an:

```
int World::getObjectID(int tile) {
    if (tile < tileCount) {
        return tile2object[tile];
    }
    return NONE;
}
```

Die Methode prüft, ob der Tile Index innerhalb des erlaubten Bereichs liegt, und gibt dann den Inhalt der `tile2object[index]`-Tabelle zurück. Diese Tabelle speichert die Objekt-ID für jede geladene Kachel. Falls ein Tile kein Objekt sein sollte, wird NONE zurückgegeben, um anzuzeigen, dass es sich nicht um ein Objekt handelt.

Die tatsächliche Zuordnung wird in `readLevel()` erstellt.

```
bool World::readLevel(int id) {

    if (map) {
        delete map;
        map = NULL;
    }
    if (tiles) {
        destroy_bitmap(tiles);
        tiles = NULL;
    }
```

```
if (tile2object){
    delete [] tile2object;
    tile2object = NULL;
}

char buffer[200];
sprintf(buffer, "level%i", id);

push_config_state();
set_config_file("game.ini");

const char *mapName  = get_config_string(
                            buffer, "map"  , NULL);
const char *tileName = get_config_string(
                            buffer, "tiles", NULL);
if (mapName == NULL || tileName == NULL) {
    pop_config_state();
    return false;
}
map = new Map();
map->load(mapName);

int objectLayer = get_config_int(
                        buffer, "objectLayer", 0);
map->setObjectLayer(objectLayer);

tiles   = load_bitmap(tileName , NULL);

tileWidth     = get_config_int(
                      tileName, "tileWidth" , 32);
tileHeight    = get_config_int(
                      tileName, "tileHeight", 32);
firstWalkable = get_config_int(
                      tileName, "walkable", 0);
map->setTiles(tiles, tileWidth, tileHeight,
              firstWalkable);

tileCount = tiles->w / tileWidth;
tile2object = new int[tileCount];

for (int a=0; a < tileCount; ++a) {
    tile2object[a] = NONE;
}

const char* list[] = {
    "key", "lockedChest", "openChest",
```

```
        "food25", "food50", "food75", "health",
        "mana", "shield", "speed",
        "damage", "door", NULL
    };

    int defaultScore = get_config_int(
                            tileName, "defaultScore", 0);
    for (int a=0; list[a]; ++a) {
        int argc;
        char**  argv;
        argv = get_config_argv(tileName,
                                list[a], &argc);
        for (int b=0; b < argc; ++b) {
            int value = atoi(argv[b]);
            if (value > 0) {
                tile2object[value] = a+1;
            }
        }
        sprintf(buffer, "%sScore", list[a]);
        scores[a+1] = get_config_int(tileName,
                                  buffer, defaultScore);
    }
    scores[0] = 0;

    pop_config_state();
    return true;
}
```

Am Anfang gibt die Methode gegebenenfalls belegten Speicher frei, um Speicherlecks zu vermeiden. Dann wird das Config File gesetzt, und der Name der Karte und der Name der Tile-Grafikdatei ausgelesen.

Falls eine dieser beiden Informationen fehlen sollte, bricht die Methode die weitere Ausführung ab, und gibt `false` zurück.

Anschließend wird die Karte geladen, und die Objektebene gesetzt.

Sobald die Tilegrafiken geladen sind, wird auch die zugehörige Metainformation zu den Kacheln aus der `game.ini` gelesen. Sobald die Größe der Tiles und der Index der ersten begehbaren Kachel bekannt ist, kann die Tileinformation der Karte gesetzt werden.

Nun wird das `tile2object` Array mit Daten gefüllt. Die Anzahl der Tiles wird aus der Größe des Bildes und der Tilegröße berechnet, und ein dementsprechend großes `int`-Array wird angelegt – und erst einmal mit 0 gefüllt.

Dann wird im Abschnitt der Tilegrafiken noch nach möglichen Objektattributen gesucht. Die Namen der Attribute stehen in einem lokalen String-Array namens `list`.

```
const char* list[] = {
        "lockedChest", "openChest", "key",
        "food25", "food50", "food75", "health",
        "mana", "shield", "attack", "speed",
        "damage", "door", NULL
    };
```

Das Array endet mit einem `Null` Wert – auf diese Weise sparen wir es uns, manuell die Anzahl der Einträge im Array mitzuführen.

Es wird in der `game.ini` nach einem Array von Werten für jeden dieser Schlüssel gesucht.

```
for (int a=0; list[a]; ++a) {
        int argc;
        char**  argv;
        argv = get_config_argv(tileName,
                                list[a], &argc);
```

Dann wird jede gelistete Eintrag (der Tile Index) mit der Objekt ID verknüpft.

```
        for (int b=0; b < argc; ++b) {
            int value = atoi(argv[b]);
            if (value > 0) {
                tile2object[value] = a+1;
            }
        }
        sprintf(buffer, "%sScore", list[a]);
        scores[a+1] = get_config_int(tileName,
                                     buffer,
                                     defaultScore);
    }
```

Ab diesem Zeitpunkt kann über `tile2object[ tileIndex]` immer der korrekte Objekttyp ermittelt werden – natürlich nur, wenn der Eintrag in der `game.ini` fehlerlos ist.

Neben der eigentlichen Zuordnung wird auch noch die Punktzahl für jedes Objekt gespeichert – der Spieler soll ja für das Aufnehmen der Objekte auch belohnt werden.

Bevor die Methode mit einem Rückgabewert von `true` ihren Erfolg signalisiert, wird noch der alte Config Stand wiederhergestellt.

Und auch das Hauptprogramm kann nun deutlich übersichtlicher gestaltet werden. Es sind keine Aufrufe der Config-Routinen mehr nötig, und so reichen ein paar Zeilen um das Spiel zu initialisieren.

```
World world;
world.readLevel(0);

Map *map = world.getMap();
map->setTargetBitmap(doubleBuffer);

hero->setMap(map);
hero->setPosition(map->getStartX() * TILE_W,
                  map->getStartY() * TILE_H);
```

Nun sind alle Voraussetzungen geschaffen, um in der Kollisionsabfrage auch den Objekttyp zu berücksichtigen.

Der Objekttyp

In der `Sprite`-Klasse wurde bisher das Objekt direkt von der Karte entfernt. Das Problem ist jetzt aber, dass Türen auch Objekte sind, diese aber nur entfernt werden dürfen, wenn auch ein Schlüssel gefunden wurde.

Aus diesem Grund erweitern wir die `objectTaken()`-Methode etwas. Sie bekommt nun die Objekt-ID des gefundenen Tiles und gibt `true` zurück, wenn das Objekt aufgenommen wurde. Gibt die Methode `false` zurück, dann konnte das Objekt nicht aufgenommen werden.

```
if (!map->isWalkable(tx, ty)) {
    x = ox;
    y = oy;
} else {
    tx = (x+ frameWidth  /2)/tw;
    ty = (y+ frameHeight /2)/th;
    int layer = map->getObjectLayer();
    int obj = world->getObjectID(
                        map->getTileAt(tx, ty, layer)
                    );
    if (obj) {
        if (objectTaken(obj)) {
            map->setTileAt(tx, ty, layer, 0);
        } else {
            x = ox;
            y = oy;
        }
    }
}
```

Die Änderungen hier sind klein, aber entscheidend. Anstatt des Tileindex wird die Objekt ID ermittelt, durch einen Aufruf von `world->getObjectID()`. Dann wird `objectTaken()` aufgerufen, um zu überprüfen, ob das Objekt auch aufgenommen werden kann. Ist dies der Fall, dann wird das Tile des Objektes von der Karte gelöscht. Wenn nicht, kann das Sprite die entsprechende Position auch nicht betreten – es wird wie eine normale Kollision behandelt.

Sie können den `world` Zeiger entweder sowohl in der `Map` als auch in der `Sprite` Klasse speichern oder eine globale Variable für die Welt anlegen. Ich habe mich für die globale Variable entschieden – aber Sie können selbstverständlich in Ihrem Spiel den Zeiger auch in den einzelnen Objekten ablegen.

Nun können Sie in Ihrer `objectTaken()`-Methode auf die unterschiedlichen Objekttypen reagieren.

```
bool PlayerCharacter::objectTaken(int id) {
    bool result = true;
    switch (id) {
        case World::DOOR:
        case World::LOCKED_CHEST:
            if (keyCount == 0) {
                result = false;
            } else {
                --keyCount;
            }
            break;

        case World::SPEED_POTION:
            setSpeed(normSpeed * 4);
            speedTimeOut = 60 * 10;
            break;

        case World::KEY:
            ++keyCount;
        break;
    }

    if (result) {
        score += world->getScore(id);

        playSound(samples[YEAH], 250);
        curBubble = YEAH;
        bubbleTimeOut = 60;
    }

    return result;
}
```

Die Methode überprüft bei Türen und verschlossenen Schatztruhen, ob der Spieler einen (oder mehrere) Schlüssel hat. Ist dies der Fall, dann kann die Schatztruhe aufgenommen beziehungsweise die Tür geöffnet werden.

Findet der Spieler einen Geschwindigkeitstrank, dann wird er für zehn Sekunden vier mal so schnell wie normal. Um dies zu realisieren, brauchen wir einige neue Attribute, darunter die normale Geschwindigkeit des Charakters und einen Zähler, um den Bonus nach Ablauf der Zeit zurückzunehmen.

Wird ein Schlüssel aufgenommen, dann wird die Anzahl der vorhandenen Schlüssel um eins erhöht.

In jedem Fall wird beim Aufnehmen eines Objektes der Punktestand erhöht.

Der Zähler für den Geschwindigkeitstrank wird in der `update()`-Methode heruntergezählt. Hier wird auch schließlich der Bonus aufgehoben.

```
void PlayerCharacter::update() {
    Sprite::update();
    if (bubbleTimeOut) {
        --bubbleTimeOut;
        if (bubbleTimeOut == 0) {
            curBubble = -1;
        }
    }
    if (speedTimeOut) {
        --speedTimeOut;
        if (speedTimeOut == 0) {
            setSpeed(normSpeed);
        }
    }
}
```

Jetzt kann der Spieler nicht nur Objekte aufsammeln, die unterschiedlichen Objekte erzeugen auch unterschiedliche Effekte. Und alle erhöhen seine Punktzahl. Türen sind kein Hindernis mehr (sofern die Schlüssel gefunden werden) – jetzt fehlen eigentlich nur noch Gegner.

Und um die Gegner kümmern wir uns im nächsten Kapitel.

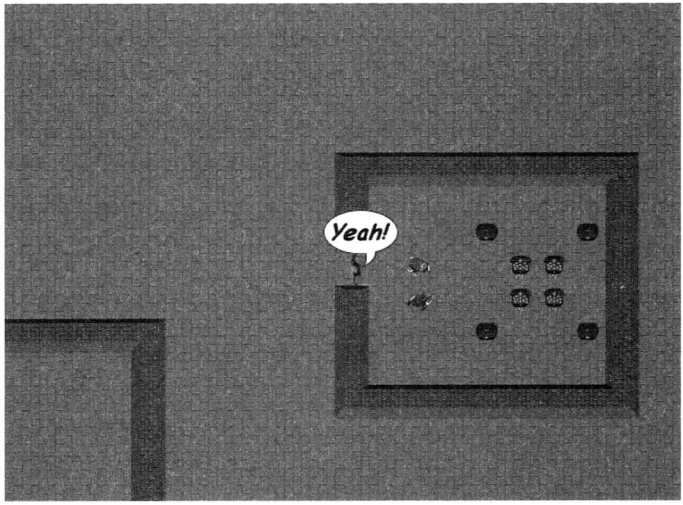


Abbildung 26.3: Der Held in der Schatzkammer

27 Gegenspieler

Bisher konnte unser Held in dem Level unbehelligt umherstreifen. In diesem Kapitel werden wir ihm ein paar Gegner auf den Hals hetzen. Am Ende dieses Kapitels sind Sie in der Lage, eigene Monster mit individuellen Eigenschaften zu programmieren.

Monster sind auch nur Sprites

Ohne Gegner kann ein Held einfach kein Held sein. Ohne Horden von Untoten fehlt dem Spiel das Hauptelement.

Da sich die Monster genauso im Kerker bewegen können sollen wie der Held, brauchen sie auch die gleiche Anzahl von Bildern für die unterschiedlichen Richtungen.

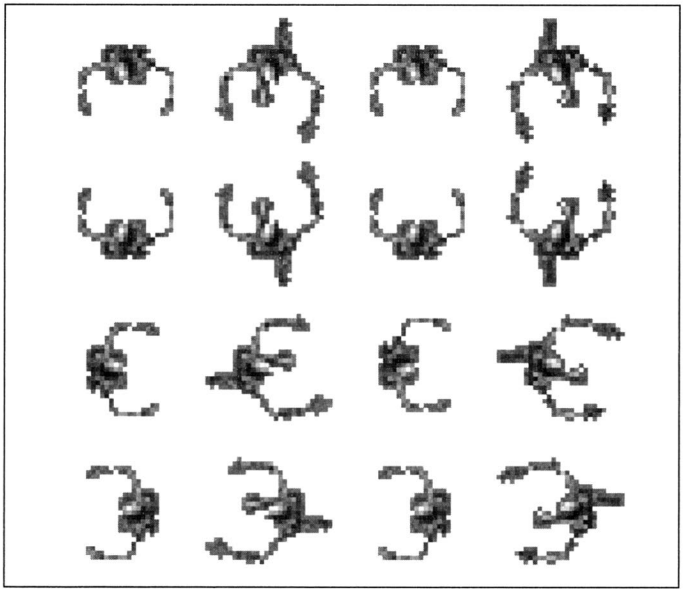


Abbildung 27.1: Die Frames für das Skelettmonster

Die Monster sollen über den Karteneditor platzierbar sein, und auch die Eigenschaften der Monster sollen auf eine einfache Weise anpassbar sein.

Eigenschaften der Monster

Jedes Monster hat (wie auch der Spieler) eine Geschwindigkeit, die angibt wie schnell es sich bewegen kann. Da es sich um ein Monster handelt, muss auch festgelegt werden, wie häufig, wie stark und mit welcher Geschicklichkeit das Monster angreift.

Auch ist es immer gut zu wissen, wie viele Treffer so ein Monster aushält und wie einfach es zu treffen ist.

Die gleichen Eigenschaften brauchen wir aber auch für unseren Helden... es wird Zeit, die Klassenstruktur leicht zu ändern.

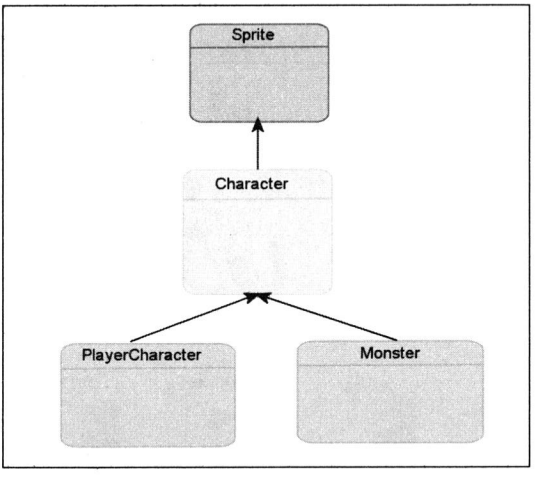


Abbildung 27.2: Die neue Klassenhierarchie

Sowohl die `PlayerCharacter`-Klasse als auch die `Monster`-Klasse sind `Character`-Klassen. Mit anderen Worten: Alle Eigenschaften und Methoden der `Character`-Klasse gelten für `Monster` und `PlayerCharacter`.

Welche Eigenschaften brauchen wir?

- ✔ Die Stärke gibt an, wie hoch der verursachte Schaden ist.
- ✔ Die Gesundheit gibt an, wie viele Treffer eingesteckt werden können.
- ✔ Die Trefferchance gibt an, wie gut der Charakter angreifen kann.
- ✔ Die Abwehrchance gibt an, wie gut der Charakter in defensiven Maßnahmen ist.
- ✔ Die Geschwindigkeit gibt an, wie schnell sich der Charakter bewegen kann.

Die einzelnen Attribute müssen gesetzt und gelesen werden können, und eine Methode alle Werte aus einem Config File zu laden, wäre auch sehr praktisch.

```
#ifndef CHARACTER_HEADER
#define CHARACTER_HEADER

#include <allegro.h>
#include "util.h"
#include "map.h"
#include "sprite.h"

class Character : public Sprite {
public:
    enum {
        VITALITY = 0,
        STRENGTH = 1,
        ATTACK   = 2,
        DEFENSE  = 3,
        SPEED    = 4,
        ATTRIBUTE_COUNT = 5,
    };

    Character() {};
    Character(BITMAP *img, int fw, int fh);
    virtual ~Character();

    virtual bool  create();
    virtual void  load(const char* section);
    virtual float getAttribute(int index);
    virtual void  setAttribute(int index, float value);
```

```
private:
    float attributes[ATTRIBUTE_COUNT];
    bool  freeImage;

};

#endif
```

Die `load()`-Methode bekommt die `section` der Konfigurationsdatei, in der die Informationen zu finden sind, als Parameter übergegeben.

```
#include "character.h"
#include "world.h"

Character::Character(BITMAP *img, int fw, int fh) : Sprite (img, fw,
fh) {
}

bool Character::create() {
    bool result = Sprite::create();

    return result;
}

Character::~Character() {
    if (freeImage) {
        destroy_bitmap(frames);
        frames = NULL;
    }
}

float Character::getAttribute(int index) {
    return attributes[index];
}
void  Character::setAttribute(int index, float value) {
    attributes[index] = value;
}


void Character::load(const char* section) {
    const char *labels[] = {
        "vitality", "strength",
        "attack"  , "defense",
        "speed"
    };
```

```
    for (int a=0; a < ATTRIBUTE_COUNT; ++a) {
        attributes[a] = get_config_float(
                                section, labels[a], 1.0);
    }

    const char *imgName = get_config_string(
                                  section, "gfx", NULL);
    if (imgName) {
        freeImage   = true;
        frames      = load_bitmap(imgName, NULL);;
        frameWidth  = get_config_int(
                          section, "frameWidth" , 32);
        frameHeight = get_config_int(
                          section, "frameHeight", 32);
        countFrames = frames->w/ frameWidth;
    }
}
```

Um das Erschaffen der Monster möglichst einfach zu gestalten, wird auch der Name der Bilddatei und die Größe eines Frames direkt in der INI-Datei angegeben.

Beim Erzeugen des Monsters wird dann auch die entsprechende Bilddatei geladen. Allerdings gibt es hier ein Problem.

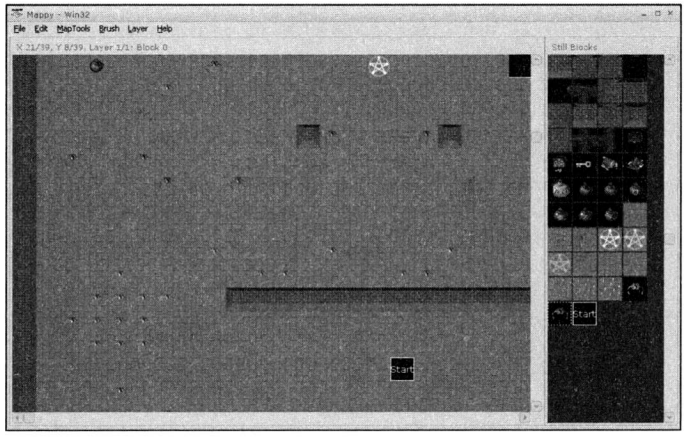


Abbildung 27.3: Die Monster im Editor

Wenn jedes der Monster die Grafikdatei lädt, dann verbrauchen wir Unmengen an Speicher – und dies unnötig. Es ist ausreichend, wenn die Grafik für jedes Monster nur einmal geladen wird. Wir brauchen eine Möglichkeit, von einem bestehenden Monster eine Kopie zu erzeugen. Der Copy Constructor, der von C++ automatisch erzeugt wird, ist beinahe das, was wir brauchen. Es gibt nur einen Unterschied: Bei den kopierten Monstern darf das `freeImage`-Attribut nicht gesetzt sein.

Deswegen bleibt uns nichts anderes übrig, als den Copy Constructor selbst zu schreiben.

```
Character::Character(const Character& rhs)
                    : Sprite(rhs) {
    this->freeImage = false;
    for (int a=0; a < ATTRIBUTE_COUNT; ++a) {
        this->setAttribute(a, rhs.getAttribute(a));
    }
}
```

Dieser Code sieht auf den ersten Blick OK aus, doch wenn wir ihn kompilieren, erhalten wir eine seltsame Fehlermeldung:

```
g++  -mwindows -D__GTHREAD_HIDE_WIN32API -Wall  -c -o character.o
character.cpp
character.cpp: In copy constructor 'Character::Character(const
Character&)':
character.cpp:16: passing 'const Character' as 'this' argument of
'virtual
   float Character::getAttribute(int)' discards qualifiers
make: *** [character.o] Error 1
```

Was will uns diese doch recht kryptische Fehlermeldung sagen? Es geht offensichtlich um einen `const Character`. Da nur der Parameter des Copy Constructors als `const` deklariert ist, wird es sich wohl um diesen handeln. Aber was ist das Problem? Wir rufen doch nur die `getAttribute()`-Methode des übergebenen Parameters auf?

Genau hier liegt das Problem. Da der Parameter `rhs` als `const` deklariert wurde, dürfen nur Methoden aufgerufen werden, die keinen Wert von `rhs` ändern. Zwar trifft dies auf die `getAttribute()`-Methode zu, aber der Compiler kann das nicht wissen. Also müssen wir es ihm sagen. Dies machen wir, indem die `getAttribute()`-Methode als `const` deklariert wird:

```
virtual float getAttribute(int index) const;
```

Und schon ist auch dieses Problem erledigt. Beim nächsten Aufruf von `make` kompiliert die Klasse ohne Probleme.

Die game.ini erweitern

Um die Monster (in diesem Fall unsere Skelette) laden zu können, muss die `game.ini` um einige Einträge erweitert werden. Die Tile-Sektion benötigt Einträge, die die Tiles mit den Monsterbildern mit den jeweiligen Monsterdefinitionen verbindet.

```
[tile.tga]
# Die Sachen hier sind bekannt
# ...
# aber die Monster Infos sind neu
monsterCount=2
monster0tile=79
monster0name=skel
monster1tile=80
monster1name=redskel

[skel]
vitality=10
strength=10
attack=1
defense=1
speed=0.05
gfx=skel.tga
frameWidth=32
frameHeight=32

[redskel]
vitality=20
strength=15
attack=2
defense=1
speed=0.12
gfx=redskel.tga
frameWidth=32
frameHeight=32
```

Das Tile File `tile.tga` enthält zwei Monster (`monsterCount=2`). Für jedes Monster wird ein Tile-Index angegeben (`monsterNindex`). Wird dieser Index in der Karte gefunden, bedeutet dies: Hier ein Monster des jeweiligen Typs platzieren.

Zusätzlich zum Tile Index wird auch der Name des Monsters angegeben (`monsterNname`). Dieser Name ist gleichzeitig der Sektionsname, unter dem die Attributsinformationen zu finden sind.

Wir haben zwei Monster definiert, ein Monster namens »skel«, welches recht langsam und ziemlich schwach auf der Brust ist, und ein Monster namens »redskel«, das schon etwas mehr auf dem Kasten hat.

Das zweite Monster nutzt die gleichen Grafiken wie das erste, allerdings hat das redskel einen deutlichen Rotstich, damit man es von einem normalen Skelett unterscheiden kann.

Die Monster laden

Beim Laden des Levels in der `World` Klasse legen wir nun für jedes der Tile File vorhandenen Monster ein Original an, von dem wir dann beliebig viele Kopien erstellen. Dafür legen wir ein Monster-Depot (engl.: Repository) an.

Neben der eigentlichen `Monster`-Klasse speichern wir auch noch das Tile, das ein Monster dieses Typs repräsentiert und die Anzahl der unterschiedlichen Monster.

```
int       monsterCount;
Monster** monsterRepository;
int*      tile2monster;
```

Die `Monster`-Klasse selbst ist im Moment noch nicht sehr ausgearbeitet:

```
// Monster.h
#ifndef MONSTER_HEADER
#define MONSTER_HEADER

#include "character.h"

class Monster : public Character {
public:
    Monster() {};
    Monster(BITMAP *img, int fw, int fh)
           : Character(img, fw, fh) {
    }

    Monster(const Monster &monster)
           : Character(monster) {};
};

#endif
```

Das `Monster` ist bisher nur ein `Character` – aber das wird sich bald ändern.

Jetzt geht es erst einmal darum, das monsterRepository mit Werten zu füllen. Dies geschieht in der readLevel()-Methode. In einem ersten Schritt wird in der Sektion des Tile Files nach dem monsterCount-Eintrag gesucht. Wird dieser gefunden, dann werden das monsterRepository und das tile2monster-Array angelegt.

```
monsterCount = get_config_int(
                        tileName, "monsterCount", 0);

if (monsterCount > 0) {
    monsterRepository = new Monster* [monsterCount];
    tile2monster      = new int[monsterCount];
} else {
    monsterRepository = NULL;
    tile2monster      = NULL;
}
```

Und sobald wir den Code schreiben, der Speicher anlegt, schreiben wir auch sofort den Code, der diesen Speicher später wieder freigibt. In unserem Fall bedeutet dies, dass wir das monsterRepository und das tile2monster-Array sowohl im Destructor als auch am Beginn der readLevel()-Methode freigeben müssen.

```
World::~World() {
    if (map) {
        delete map;
        map = NULL;
    }
    if (tiles) {
        destroy_bitmap(tiles);
        tiles = NULL;
    }
    if (tile2object){
        delete [] tile2object;
        tile2object = NULL;
    }
    if (monsterRepository) {
        for (int a=0; a < monsterCount; ++a) {
            if (monsterRepository[a]) {
                delete monsterRepository[a];
            }
        }
        delete [] monsterRepository;
    }
    if (tile2monster) {
        delete [] tile2monster;
    }
}
```

Beim monsterRepository muss beachtet werden, dass sowohl die Objekte im Array als auch das Array selbst freigegeben werden, da ein delete [] bei einem Array von Zeigern nie die Objekte freigibt, auf die gezeigt wird, sondern nur den von den Zeigern belegten Speicher.

Die gleichen Aufrufe stehen auch zu Beginn der readLevel()-Methode.

```
bool World::readLevel(int id) {

    if (map) {
        delete map;
        map = NULL;
    }
    if (tiles) {
        destroy_bitmap(tiles);
        tiles = NULL;
    }
    if (tile2object){
        delete [] tile2object;
        tile2object = NULL;
    }
    if (monsterRepository) {
        for (int a=0; a < monsterCount; ++a) {
            if (monsterRepository[a]) {
                delete monsterRepository[a];
            }
        }
        delete [] monsterRepository;
    }
    // ...
    // Und hier der Rest von readLevel
    // ...
}
```

Da nun das Monsterdepot bereitsteht können wir es mit Bewohnern füllen. Für jedes angegebene Monster suchen wir den Namen und den Tile Index. Sobald wir diese beiden Werte haben, können wir das Monster mit der load()-Methode initialisieren, und im tile2monster-Array den korrespondierenden Tile Index setzen.

```
for (int a=0; a < monsterCount; ++a) {
    sprintf(buffer, "monster%itile", a);
    int index = get_config_int(tileName, buffer, -1);
    if (index >0) {
        sprintf(buffer, "monster%iname", a);
        const char *monsterName
```

```
                        = get_config_string( tileName,
                                             buffer,
                                             NULL);
        if (monsterName) {
            Monster *monster = new Monster();
            monster->load(monsterName);
            tile2monster[a] = index;
            monsterRepository[a] = monster;
        } else {
            tile2monster[a] = -1;
        }
    }
}
```

In einem letzten Schritt müssen wir nun noch die `Map` nach Vorkommen der Monster Tiles durchsuchen und für jedes gefundene Tile ein neues Monster anlegen bzw. das Monster aus dem `monsterRepository` kopieren. Dieses neue Monster wird dann in einer STL-Liste gespeichert.

```
// character.h
typedef std::list<Character*> CharacterList;
typedef CharacterList::iterator CharacterItor;

// in world.h
CharacterList *characterList;
```

Wir durchsuchen die komplette Karte, und wenn wir auf der Objektebene ein Monster-Tile finden, dann wird ein neues Monster an dieser Stelle erzeugt und das Monster Tile von der Karte entfernt.

```
for (int y=0; y < map->getHeight(); ++y) {
    for (int x=0; x < map->getWidth(); ++x) {
        int tile = map->getTileAt(x,y,objectLayer);
        for (int a=0; a < monsterCount; ++a) {
            if (tile2monster[a] == tile) {
                map->setTileAt(x,y,objectLayer, 0);
                Monster *monster = new Monster(
                                *monsterRepository[a]);
                monster->setPosition(x * tileWidth,
                                     y * tileHeight);
                characterList->push_back(monster);
            }
        }
    }
}
```

Natürlich muss auch die `characterList` freigegeben werden. Entsprechende `delete`-Aufrufe stehen im Destructor und der `readLevel()`-Methode.

Wenn wir nun das Programm kompilieren und starten, dauert es etwas länger, und dann sehen wir den alt bekannten Level. Die Monster sind nicht zu sehen, da wir haben die Monster zwar haben, allerdings kein Code vorhanden ist, der diese auch anzeigt.

Diesen Code fügen wir direkt ins Hauptprogramm ein:

```
if (needsRedraw) {
    int x = MID(0,
                hero->getX() - SCREEN_W/2,
                map->getWidth() * TILE_W - SCREEN_W);
    int y = MID(0,
                hero->getY() - SCREEN_H/2,
                map->getHeight()* TILE_H - SCREEN_H);
    map->draw(x, y);

    CharacterItor itor = world->getCharacters()->
                                            begin();
    CharacterItor end  = world->getCharacters()->end();
    int count = 0;
    for (;itor != end; ++itor) {
        Character* c = *itor;
        c->draw(doubleBuffer, x, y);
        ++count;
    }
    hero->draw(doubleBuffer, x,y);
    needsRedraw = false;
    show();
}
```

Und schon tummeln sich Dutzende von Skeletten im Dungeon.

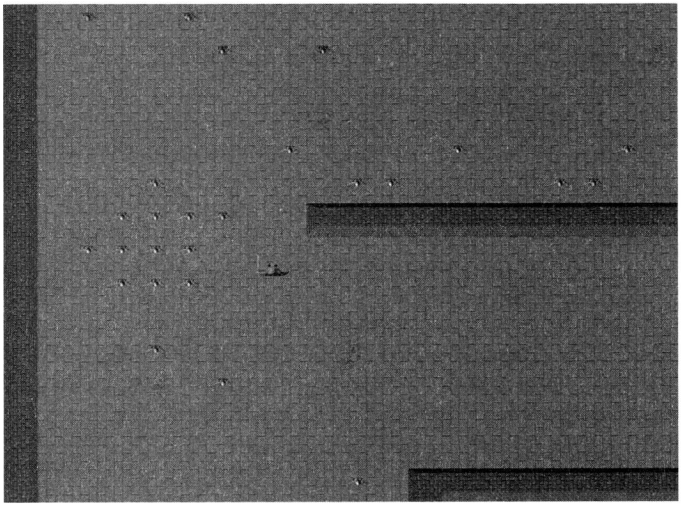


Abbildung 27.4: Lauter Skelette

Künstliche Intelligenz

Bisher stehen die Skelette nur starr auf ihrem Platz und nehmen keine Notiz von dem Spieler. Dies werden wir nun ändern. Der erste Schritt in Richtung Selbstbestimmung für die Monster ist der Aufruf der `update()`-Funktion jedes Monsters im Hauptprogramm.

```
if (timerCounter) {
    joypad->poll();

    // Spieler Steuerung hier

    do {
        hero->update();
       // Monster updaten
        CharacterItor itor = world->getCharacters()->
                                                begin();
        CharacterItor end  = world->getCharacters()->
                                                  end();
```

```
        int count = 0;
        for (;itor != end; ++itor) {
            Character* c = *itor;
            c->update();
            ++count;
        }
        --timerCounter;
    } while (timerCounter >0);
    needsRedraw = true;
}
```

Nun wird die `update()`-Methode aufgerufen. Die Skelette sollen auf den Spieler zulaufen. In einem ersten Schritt lassen wir die Skelette sich in die Richtung des Spielers drehen:

```
void Monster::update() {

    PlayerCharacter *hero = world->getHero();
    if (!hero) {
        return;
    }
    int dx = hero->getX() - getX();
    int dy = hero->getY() - getY();
    if (ABS(dx) > ABS(dy)) {
        if (dx < 0) {
            curDirection  = LEFT;
        } else {
            curDirection  = RIGHT;
        }
    } else {
        if (dy < 0) {
            curDirection  = UP;
        } else {
            curDirection  = DOWN;
        }
    }
}
```

»Moment!«, werden Sie sich jetzt denken, »seit wann hat denn die `World`-Klasse eine `getHero()`-Methode?«

Guter Punkt. Bis jetzt hat sie diese Methode noch nicht, aber wenn die Skelette wissen sollen, wo der Spieler ist, dann brauchen sie diese Methode. Also fügen wir sie zur `World`-Klasse hinzu, deren Header File inzwischen wie folgt aussieht:

```
#ifndef WORLD_HEADER
#define WORLD_HEADER

#include <allegro.h>
#include "map.h"
#include "monster.h"
#include "playercharacter.h"

class World {

public:
    enum {
        NONE            = 0,
        KEY             = 1,
        LOCKED_CHEST    = 2,
        OPEN_CHEST      = 3,
        FOOD25          = 4,
        FOOD50          = 5,
        FOOD75          = 6,
        HEALTH_POTION   = 7,
        MANA_POTION     = 8,
        SHIELD_POTION   = 9,
        SPEED_POTION    =10,
        STRENGTH_POTION =11,
        DOOR            =12,
    };
    enum {
        OBJECT_COUNT = 13
    };

    World();
    ~World();

    int getObjectID(int tile);
    bool readLevel(int id);

    Map* getMap();
    BITMAP *getTiles();
    int getFirstWalkable();
    int getScore(int objectID);

    CharacterList *getCharacters();
    PlayerCharacter *getHero();
    void setHero(PlayerCharacter *hero);
```

```
private:
    int  scores[OBJECT_COUNT];
    int  tileCount;
    int* tile2object;
    BITMAP *tiles;
    Map    *map;

    int tileWidth;
    int tileHeight;
    int firstWalkable;

    int       monsterCount;
    Monster** monsterRepository;
    int*      tile2monster;

    CharacterList *characterList;
    PlayerCharacter *hero;
};

extern World *world;

#endif
```

Im Hauptprogramm wird dann die Variable gesetzt:

```
PlayerCharacter *hero = new PlayerCharacter(heroSpr,
                                            TILE_W, TILE_H);

world = new World();
world->readLevel(0);
world->setHero(hero);
```

Wenn Sie das Programm jetzt starten, drehen sich die Skelette in die Richtung des Spielers.

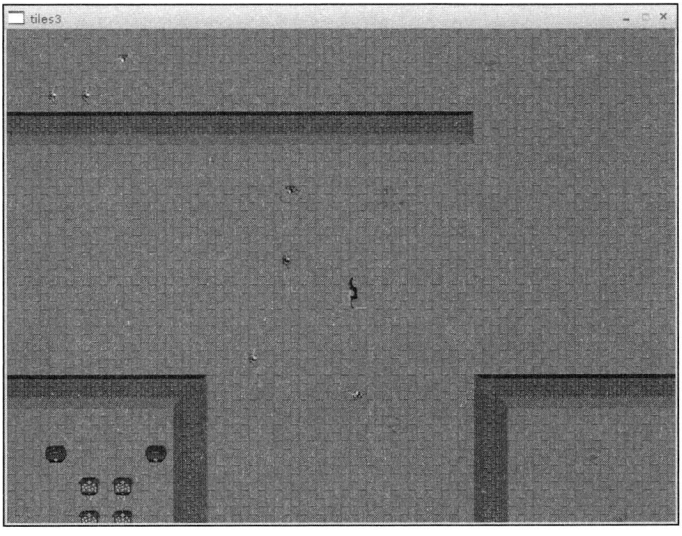


Abbildung 27.5: Die Skelette drehen sich in Richtung des Spielers

Wenn wir den Skeletten nun auch noch sagen, sie sollen sich in Bewegung setzen, dann haben wir in Sekundenschnelle einen Haufen von Skeletten um unseren Helden herum.

```
void Monster::update() {
    PlayerCharacter *hero = world->getHero();
    if (!hero) {
        return;
    }

    int dx = hero->getX() - getX();
    int dy = hero->getY() - getY();

    // let's see if the sprite is too far
    // from the player
    if (dx*dx +dy *dy > SCREEN_W*SCREEN_H*4) {
        return;
    }

    if (ABS(dx) > ABS(dy) && !minor) {
```

```
        if (dx < 0) {
            curDirection  = LEFT;
        } else {
            curDirection  = RIGHT;
        }
    } else {
        if (dy < 0) {
            curDirection  = UP;
        } else {
            curDirection  = DOWN;
        }
    }
    moving = true;
    Character::update();
}
```

Allerdings rennen die Skelette noch immer wild durcheinander – und dies im wahrsten Sinne des Wortes, da es noch keine Kollisionsabfrage zwischen den Sprites gibt.

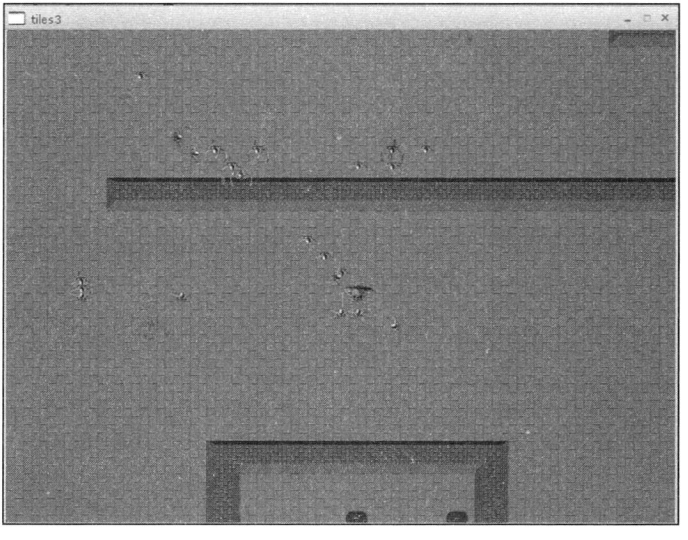


Abbildung 27.6: Auf den Spieler zustürmende Sprites

Da kein Sprite in ein anderes hineinlaufen können soll, ändern wir die update()-Methode der Sprite-Klasse direkt.

```
void Sprite::update() {
    if (isMoving()) {
        updateCounter += speed;
        while (updateCounter >= 1.0) {

            incFrame(1);
            int ox = x;
            int oy = y;
            switch (curDirection) {
                case DOWN:
                    x += deltaX[curFrame];
                    y += deltaY[curFrame];
                break;

                case UP:
                    // Y negieren
                    x += deltaX[curFrame];
                    y -= deltaY[curFrame];
                break;

                case RIGHT:
                    // X und Y vertauschen
                    x += deltaY[curFrame];
                    y += deltaX[curFrame];
                break;

                case LEFT:
                    // X und Y vertauschen
                    // X negieren
                    x -= deltaY[curFrame];
                    y += deltaX[curFrame];
                break;
            }
            if (map) {
                int tw = map->getTileWidth();
                int th = map->getTileHeight();

                int tx = x / tw;
                int ty = y / th;

                if (curDirection == RIGHT) {
                    tx++;
```

```
} else if (curDirection == DOWN) {
    ty++;
}

if (!map->isWalkable(tx, ty)) {
    x = ox;
    y = oy;
    moving = false;
} else {
    tx = (x+ frameWidth  /2)/tw;
    ty = (y+ frameHeight /2)/th;
    int layer = map->getObjectLayer();
    int obj = world->getObjectID(map->getTileAt(tx,
  ty, layer));

    if (obj) {
        if (objectTaken(obj)) {
            map->setTileAt(tx, ty, layer, 0);
        } else {
            x = ox;
            y = oy;
            moving = false;
        }
    }
}
bool searching = true;
if (moving) {
    CharacterItor itor = world->getCharacters()-
  >begin();
    CharacterItor end  = world->getCharacters()-
  >end();
    for (;itor != end && searching; ++itor) {
        Character* c = *itor;
        // Kollision mit sich selbst?
        if (this != c) {
            if (collidesWith(c)) {
                x = ox;
                y = oy;

                searching = false;
                moving = false;
            }
        }
    }
}
```

```
            } else {
                map = world->getMap();
            }
            updateCounter -= 1.0;
        }
    } else {
        updateCounter = 0.0;
    }
}

bool Sprite::collidesWith(Sprite* spr) {
    if (x + frameWidth <= spr->x ||
        x >= spr->x + spr->frameWidth ||
        y + frameHeight <= spr->y ||
        y >= spr->y + spr->frameHeight) {
            return false;
    }
    return true;
}
```

Neben der Tatsache, dass nun Sprites auf Kollisionen gecheckt werden, gibt es noch eine weitere Neuerung: Wenn eine Kollision festgestellt wird, dann wird die Variable `moving` auf `false` gesetzt.

Dies können sich die abgeleiteten Klassen zu Nutze machen, um das Verhalten des Sprites bei einer Kollision zu beeinflussen.

In den meisten Fällen werden die Monster wohl mit anderen Monstern und nicht mit dem Spielersprite kollidieren. Da die Monster allerdings die gleiche Richtung haben (auf den Spieler zu), kann es passieren, dass zwei Sprites aneinander hängen bleiben.

Dies kann man verhindern, indem man die Priorität der Bewegungsachsen ändert. Normalerweise versucht das Monster sich entlang der Achse zu bewegen, bei der die Distanz größer ist. Ist der Spieler also weiter entlang der X-Achse entfernt, dann versucht das Monster den horizontalen Abstand zu verkürzen. Ist die Entfernung zum Spieler entlang der Y-Achse größer, so wird erst dieser Abstand verringert. Bei einer Kollision könnte man nun ein Flag setzen, das dieses Verhalten umdreht. Dann würde das Monster zuerst versuchen, sich auf der kürzen der beiden Achsen zu nähern.

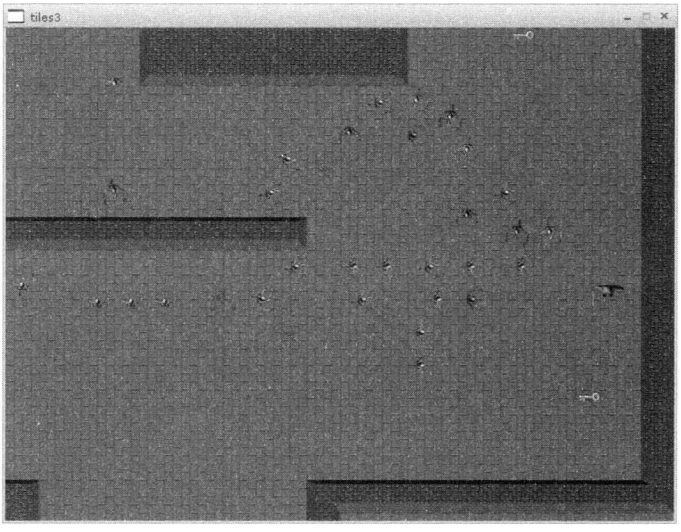


Abbildung 27.7: Spritekollisionen

```
if (ABS(dx) > ABS(dy) && !minor) {
    if (dx < 0) {
        curDirection  = LEFT;
    } else {
        curDirection  = RIGHT;
    }
} else {
    if (dy < 0) {
        curDirection  = UP;
    } else {
        curDirection  = DOWN;
    }
}
```

Wenn das Flag `minor` gesetzt ist, dann nähert sich das Monster dem Spiel nun entlang der Achse mit dem geringeren Abstand. Das Flag wird nach dem Aufruf der `update()` Methode der Basisklasse gesetzt:

```
moving = true;
Character::update();
```

```
if (!moving) {
    if (minor) {
        minor = 0;
    } else {
        minor = 1;
    }
}
```

Das hilft schon etwas, aber noch immer bleiben die Skelette recht häufig aneinander hängen. Wenn man nun jedem Monster nach einer Kollision eine kurze, zufällig lange Wartezeit gibt, dann wird dadurch die Gefahr verringert, dass zwei Monster immer wieder versuchen auf das gleiche Feld zu gelangen und sich dabei gegenseitig blockieren.

Diesen Effekt kann man noch verstärken, wenn man jedem Monster einen `IQ` gibt, der festlegt wie häufig das Monster versucht eine neue Richtung zu finden. Dieser `IQ` ist ein Attribut wie die anderen auch und sollte deswegen mittels der `Character::load()`-Methode geladen werden.

Es müssen also zwei Klassen geändert werden: Zuerst die `Character`-Klasse, um das zusätzliche Attribut bereitzustellen, und dann die `Monster`-Klasse, die dieses Attribut nutzt (und auch noch die kleine, zufällige Wartezeit nach einer Kollision ergänzt).

```
// Character.h
#ifndef CHARACTER_HEADER
#define CHARACTER_HEADER

#include <allegro.h>
#include <list>

#include "util.h"
#include "map.h"
#include "sprite.h"

class Character : public Sprite {
public:
    enum {
        VITALITY = 0,
        STRENGTH = 1,
        ATTACK   = 2,
        DEFENSE  = 3,
        SPEED    = 4,
        IQ       = 5,
        ATTRIBUTE_COUNT = 6,
```

```
    };

    Character();
    Character(BITMAP *img, int fw, int fh);
    Character(const Character& rhs);
    virtual ~Character();

    virtual void  update();
    virtual bool  create();
    virtual void  load(const char* section);
    virtual float getAttribute(int index) const;
    virtual void  setAttribute(int index, float value);

protected:
    float attributes[ATTRIBUTE_COUNT];
    bool  freeImage;
    int   iqCounter;
};

typedef std::list<Character*> CharacterList;
typedef CharacterList::iterator CharacterItor;

#endif

// Character.cpp
void Character::load(const char* section) {
    const char *labels[] = {
        "vitality", "strength",
        "attack"  , "defense",
        "speed", "iq",
    };
    for (int a=0; a < ATTRIBUTE_COUNT; ++a) {
        attributes[a] = get_config_float(section, labels[a], 1.0);
    }
    setSpeed(attributes[SPEED]);

    const char *imgName = get_config_string(
                                section, "gfx", NULL);
    if (imgName) {
        freeImage   = true;
        frames      = load_bitmap(imgName, NULL);
        frameWidth  = get_config_int(section,
                                "frameWidth" , 32);
        frameHeight = get_config_int(section,
                                "frameHeight", 32);
```

```
        countFrames = frames->w/ frameWidth;

        create();
    }
}
```

In der Monster-Klasse wird nun noch die waitAI-Variable ergänzt:

```
#ifndef MONSTER_HEADER
#define MONSTER_HEADER

#include "character.h"

class Monster : public Character {
public:
    Monster();
    Monster(BITMAP *img, int fw, int fh);

    Monster(const Monster &monster);

    void update();
    void draw(BITMAP *dest, int ox, int oy);
protected:
    int minor;
    int waitAI;
};


#endif
```

Die `update()`-Methode des Monsters überprüft nun nur noch bei jedem `Iqten`-Aufruf die Position des Spielers und geht bei einer Kollision in einen kurzen Winterschlaf:

```
void Monster::update() {
    if (waitAI > 0) {
        waitAI--;
        return;
    }
    PlayerCharacter *hero = world->getHero();
    if (!hero) {
        return;
    }
```

```
    iqCounter--;
    if (iqCounter <= 0) {
        iqCounter = (int)attributes[IQ];


        int dx = hero->getX() - getX();
        int dy = hero->getY() - getY();

        // let's see if the sprite is too far
        // from the player
        if (dx*dx +dy *dy > SCREEN_W*SCREEN_H*4) {
            return;
        }

        if (ABS(dx) > ABS(dy) && !minor) {
            if (dx < 0) {
                curDirection  = LEFT;
            } else {
                curDirection  = RIGHT;
            }
        } else {
            if (dy < 0) {
                curDirection  = UP;
            } else {
                curDirection  = DOWN;
            }
        }
    }
    moving = true;

    Character::update();

    if (!moving) {
        waitAI = rnd(50);
        iqCounter = 0;
        if (minor) {
            minor = 0;
        } else {
            minor = 1;
        }
    }
}
```

Das Ergebnis ist noch nicht 100% perfekt, aber für ein paar `if`-Abfragen schon recht ordentlich.

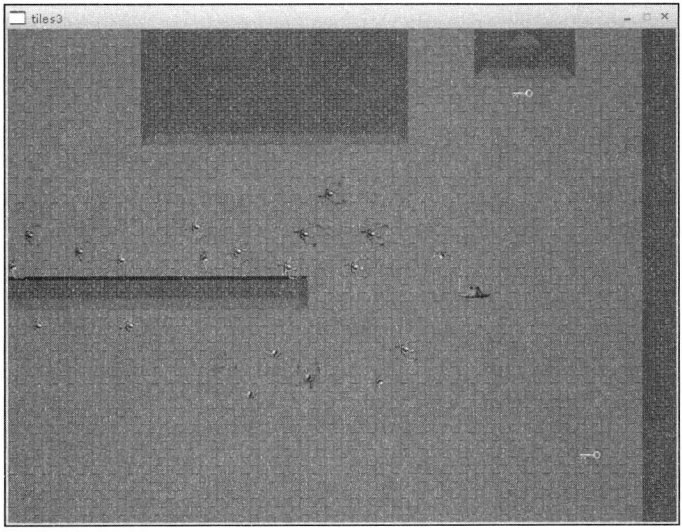


Abbildung 27.8: Die nicht mehr ganz so planlosen Skelette

28 Kampf

Bisher folgen und umkreisen die Gegner den Helden des Spieles. Vor den Gegnern davonzulaufen kann auch Spaß machen, das hat Pac-Man eindrucksvoll bewiesen. Allerdings ist ein vor den Monsterscharen davonrennender Held nicht wirklich heroisch. In diesem Kapitel erweitern wir die Spieler- und Gegnerklassen so, dass die Monster den Spieler attackieren – und natürlich geben wir dem Helden auch eine Möglichkeit zur Verteidigung.

Kampfregeln

Wann trifft ein Angriff? Sobald der Angreifer es schafft, die Verteidigung des Gegners zu durchdringen.

In Rollenspielen kann es sein, dass der Held einen perfekt gezielten Schlag gegen ein Monster ausführt, der Angriff aber trotzdem nicht trifft. Gerade in Action-Rollenspielen kann dies eine sehr störende Erfahrung sein, da hier neben den Werten des Charakters auch die Reaktion des Spielers eine große Rolle spielt.

Ein Kompromiss ist es, die Höhe des Schadens von dem Verhältnis von Angriff und Abwehr abhängig zu machen. Dann behält ein hoher Verteidigungswert seine Bedeutung, ohne jedoch dem Spieler das Gefühl der Kontrolle zu rauben.

Die Kampfregeln definieren wir in einer Klasse `Rules`. Alle Methoden dieser Klasse sind statisch, das heißt sie können aufgerufen werden, ohne dass eine Instanz der Klasse benötigt wird.

```
#ifndef RULES_HEADER
#define RULES_HEADER

#include "character.h"

class Rules {
public:
    static void fight(Character *attacker,
                      Character *defender);

};
```

```
#endif

#include "rules.h"

void Rules::fight(Character *attacker,  Character *defender) {
    if (attacker == NULL || defender == NULL) {
        return;
    }
    float chanceToHit = 2.0;


    if (defender->getAttribute(Character::DEFENSE) > 0) {
        chanceToHit = attacker->
                          getAttribute(Character::ATTACK)
                      / defender->
                        getAttribute(Character::DEFENSE);
    }
    float damage  = attacker->
                        getAttribute(Character::STRENGTH)
                       * chanceToHit;

    if (damage >= 1.0) {
        float vitality = defender->
                       getAttribute(Character::VITALITY);
        vitality -= damage;
        if (vitality <= 0.0) {
            vitality = 0;
            defender->setAlive(false);
        }
        defender->setAttribute(
                          Character::VITALITY, vitality);
    }
}
```

Der verursachte Schaden ergibt sich aus der Stärke des Angreifers und wie hoch sein Angriffswert im Vergleich zur Verteidigung seines Gegners ist.

Wenn ein Angreifer erfolgreich Schaden verursacht (`damage >= 1.0`), dann wird überprüft, ob der erzielte Schaden reicht, den Verteidiger auszuschalten. Wenn dies der Fall ist, wird er mit `setAlive(false)` aus dem Verkehr gezogen und sicher gestellt, dass der `VITALITY` Wert nie unter 0.0 sinken kann. Am Ende wird der neue `VITALITY` Wert des Verteidigers gesetzt.

Attacke!

Spendieren wir unserem Helden eine Angriffsanimation. Er reißt sein Schwert hoch über den Kopf und nutzt dann beide Hände, um das Schwert nach unten zu ziehen.

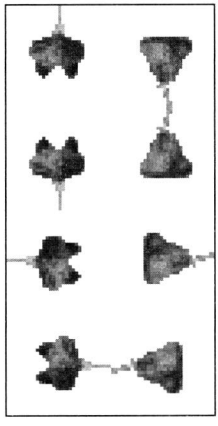


Abbildung 28.1: Die Angriffsanimation

Damit die Attacke auch abgespielt wird, sind ein paar kleine Änderungen an der `PlayerCharacter`-Klasse nötig.

Zuerst muss eine Bitmap für die Angriffsanimation hinzugefügt werden. Zusätzlich benötigen wir noch die Anzahl der Frames und ob eine boolsche Variable, die angibt, ob die Angriffsanimation gerade aktiv ist oder nicht.

```
// in PlayerCharacter.h, protected section
BITMAP *attackFrames;
int     countAttackFrames;
bool    attacking;
```

Damit diese Werte auch benutzt werden können, werden die üblichen set- und get-Methoden hinzugefügt.

```
void PlayerCharacter::setAttackFrames(BITMAP* bmp) {
    attackFrames = bmp;
    countAttackFrames = bmp->w / frameWidth;
}
```

```
BITMAP *PlayerCharacter::getAttackFrames() {
    return attackFrames;
}

void PlayerCharacter::attack() {
    moving = true;
    attacking = true;
}


bool PlayerCharacter::isAttacking() {
    return attacking;
}
```

Da die Sprite-Klasse keine Ahnung von Angriffsanimationen hat, müssen die update()- und draw()-Methoden überladen werden. In update() wird zuerst die Methode der Basisklasse aufgerufen. Danach wird im Falle eines Angriffs überprüft, ob der Spieler angehalten hat. Ist dies der Fall, hat eine Kollision stattgefunden, und wir versuchen Schaden zu verursachen.

```
void PlayerCharacter::update() {
    Character::update();
    if (attacking) {
        if (!moving) {
            Rules::fight(this,
                (Character*)lastCollision);
        }
        moving = true;
    }
    if (bubbleTimeOut) {
        --bubbleTimeOut;
        if (bubbleTimeOut == 0) {
            curBubble = -1;
        }
    }
    if (speedTimeOut) {
        --speedTimeOut;
        if (speedTimeOut == 0) {
            setSpeed(normSpeed);
        }
    }
}
```

Die draw()-Methode kümmert sich um die korrekte Darstellung des Spielersprites auf dem Bildschirm. Befinden wir uns beim Aufruf von draw() in der Kampfanimation, dann wird anstelle der normalen Bitmap die Angriffsanimation auf den Schirm geblittet.

```
void PlayerCharacter::draw(BITMAP *dest,
                                    int ox, int oy) {

    if (attacking) {
        masked_blit(attackFrames, dest,
                curFrame * frameWidth,
                curDirection * frameHeight,
                x-ox, y-oy,
                frameWidth, frameHeight);
    } else {
        Character::draw(dest, ox, oy);
    }
    if (curBubble >= 0) {
        draw_sprite(dest, bubbles[curBubble],
                    x-ox + frameWidth/2  -
                       bubbles[curBubble]->w/2,
                    y-oy + frameHeight/2-
                       bubbles[curBubble]->h);
    }
}
```

Zu guter Letzt muss von der Angriffsanimation auch wieder zurück zum Laufen gewechselt werden.

```
void PlayerCharacter::incFrame(int delta) {
    if (attacking) {
        if (curFrame == 0) {
            playSound(samples[ATTACK], 180);
        }
        curFrame += delta;
        if (curFrame >= countAttackFrames) {
            curFrame = 0;
            attacking = false;
        } else if (curFrame < 0) {
            curFrame +=countAttackFrames;
        }
    } else {
        Character::incFrame(delta);
    }
    if ((curFrame & 1)==1) {
        playSound(samples[FOOTSTEPS], 180);
    }
}
```

Damit der Spieler auch akustisch seinen Angriff miterleben kann, wird ein Sound abgespielt, sobald er zum Schlag ausholt.

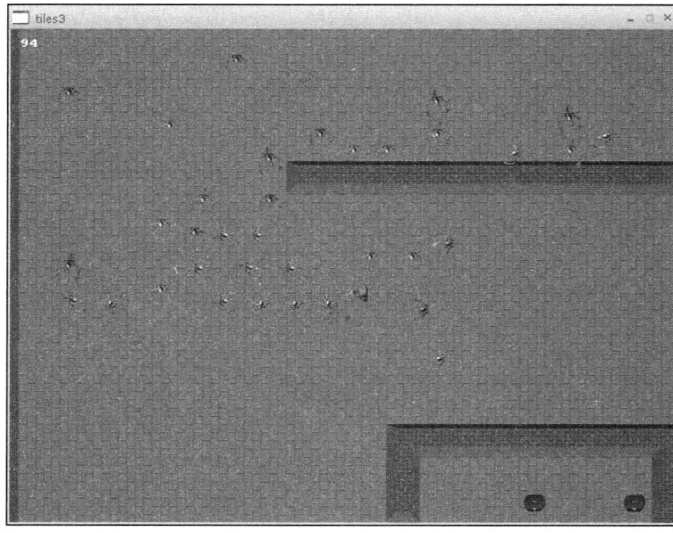


Abbildung 28.2: Der Held hackt sich durch die Massen der Gegner

Es gibt allerdings noch ein kleines Problem. Bisher hat ein `PlayerCharacter` zwar theoretisch alle Attribute, die ein `Character` auch hat, allerdings sind diese Werte bisher nicht in der `game.ini` aufgeführt – und der Spielercharakter wird auch nicht dynamisch erzeugt, sondern bisher noch fest angelegt.

Erweitern wir also die `game.ini` um einen Eintrag für den Helden:

```
[hero]
vitality=100
strength=10
attack=1
defense=1
speed=0.2
gfx=hero.tga
attackFrames=attack.tga
frameWidth=32
frameHeight=32
```

Natürlich erhält unser Held deutlich bessere Ausgangswerte als die Gegner. Die Gegner haben dafür den Mengenvorteil. Wir führen als neues At-

tribut auch die Bitmap mit dem Angriff an, damit der komplette PlayerCharacter aus der game.ini heraus erzeugt werden kann.

Die load()-Methode ist schnell erweitert. Im nächsten Schritt müssen dann auch die World-Klasse und das Hauptprogramm angepasst werden.

```
void PlayerCharacter::load(const char* section) {
    Character::load(section);
    const char* filename = get_config_string(
                        section, "attackFrames", NULL);
    if (filename == NULL) {
        return;
    }
    BITMAP *bmp = load_bitmap(filename, NULL);
    setAttackFrames(bmp);
}
```

Die Hauptarbeit erledigt die load()-Methode der Character-Klasse. In der PlayerCharacter-Methode muss nur die Bitmap geladen und gesetzt werden.

Die World-Klasse wird um eine Methode loadHero() erweitert. Auch liegt nun die gesamte Verantwortung bezüglich der Verwaltung des Spielercharakters in den Händen dieser Klasse.

```
PlayerCharacter *World::getHero() {
    if (hero == NULL) {
        loadHero("hero");
    }
    return hero;
}

void World::loadHero(const char* section) {
    if (hero) {
        getCharacters()->remove(hero);
        delete hero;
        hero = NULL;
    }

    push_config_state();
    set_config_file("game.ini");
    hero = new PlayerCharacter();
    hero->load(section);
    pop_config_state();

    getCharacters()->push_back(hero);
}
```

Falls die getHero()-Methode zu einem Zeitpunkt aufgerufen werden sollte, zu dem es noch keinen Helden gibt, wird einer erzeugt. Dieser wird dann zu der Liste mit Sprites hinzugefügt. Normalerweise wird der Held spätestens beim Laden des Levels angelegt:

```
bool World::readLevel(int id) {
    //
    // ...
    //
    // Helden auf Start setzen
    PlayerCharacter *hero = getHero();
    hero->setMap(map);
    hero->setPosition(map->getStartX() * tileWidth,
                      map->getStartY() * tileHeight);

    pop_config_state();
    return true;
}
```

Dadurch kann die gesamte Verwaltung des Helden aus dem Hauptprogramm herausgenommen werden.

```
int main(int , char**) {
    init(640, 480, 60, false);

    joypad = new Joypad();
    world = new World();
    world->readLevel(0);

    Map *map = world->getMap();
    map->setTargetBitmap(doubleBuffer);
    bool needsRedraw = false;

    // ... main loop ...

    return 0;
}
```

Nun kann zwar der Held die Monster angreifen, diese können sich aber noch nicht zur Wehr setzen. Dies ändern wir in zwei Schritten. Im ersten Schritt ändern wir die Kollisionsabfrage der Sprite-Klasse so ab, dass sie sich merkt, womit sie zusammengestoßen ist.

Und da die Chancen recht hoch sind, dass ein Monster mit einem anderen Monster zusammenstößt, überprüfen wir am Ende der Routine noch einmal gesondert, ob eine Kollision mit dem Spieler vorhanden ist.

```
bool searching = true;
if (moving) {
    CharacterItor itor = world->getCharacters()->
                                                begin();
    CharacterItor end  = world->getCharacters()->end();
    for (;itor != end && searching; ++itor) {
        Character* c = *itor;
        // Kollision mit sich selbst?
        if (this != c) {
            if (collidesWith(c)) {
                x = ox;
                y = oy;

                searching = false;
                moving = false;

                lastCollision = c;
            }
        }
    }
    // Kollision mit spieler?
    if (this != world->getHero()) {
        if (collidesWith(world->getHero())) {
            lastCollision = world->getHero();
        }
    }
}
```

Mit diesen Informationen kann dann die Monsterklasse den Spieler angreifen:

```
void Monster::update() {

    // ... Yadda - Yadda- Yadda
    moving = true;

    Character::update();

    if (!moving) {
        if (lastCollision == hero) {
            Rules::fight(this, hero);
            waitAI = rnd(500);
        } else {
            waitAI = rnd(50);
        }
        iqCounter = 0;
```

```
        if (minor) {
            minor = 0;
        } else {
            minor = 1;
        }
    }
}
```

Damit die Monster nicht konstant an dem Spieler nagen, legen sie nach einem Treffer erst einmal eine halbe Sekunde Pause ein.

Trefferfeedback

Da der Spieler nun getroffen werden kann, sollte dies dem Spieler auch deutlich gemacht werden. Ein lautes »Autsch!« passt leider nicht ganz zum typischen Barbarenimage – also muss wohl wieder das typische »Schmerzgrunzen« zum Einsatz kommen.

Zusätzlich könnte der Held auch einen markigen Spruch ablassen, je nachdem wie es ihm derzeit gesundheitlich geht.

Art des Treffers	Spruch
Normaler Treffer	Hrumpf!
50% Energie übrig	Nur ein Kratzer!
25% Energie übrig	Und jetzt etwas zu Essen!
10% Energie übrig	Das könnte knapp werden...
Stirbt	Sagen wir – unentschieden?

Tabelle 28.1: Schaden und zugehörige Sounddatei

Diese Sounds werden in der `create()`-Methode geladen und bei Änderung des `VITALITY`-Attributes abgespielt.

Zuvor wird noch die `Character`-Klasse so erweitert, dass zu jedem Attribut auch der derzeit mögliche Höchstwert gespeichert wird.

```
void  PlayerCharacter::setAttribute(
```

```
                                int index, float value) {
    float oldValue = getAttribute(index);
    Character::setAttribute(index, value);
    if (index == VITALITY && oldValue > value) {
        int percent = 100 -
                    (int) (100.0 *(attributes[VITALITY]
                          / maxAttributes[VITALITY]));
        if (!isAlive()) {
            if (percentPlayed < 100) {
                percentPlayed = 100;
                playSound(samples[DEATH], 220);
            }
        } else if (percent >=50
                  && percent > percentPlayed) {
            if (percent >= 90) {
                percentPlayed = 100;
                playSound(samples[DAM90], 220);
            } else if (percent >= 75) {
                percentPlayed = 89;
                playSound(samples[DAM75], 220);
            } else  {
                percentPlayed = 74;
                playSound(samples[DAM50], 220);
            }
        } else {
            playSound(samples[GRUNT], 220);
        }
    } else {
        percentPlayed = 0;
    }
}
```

Damit der gleiche Sound nicht immer wieder abgespielt wird, wird in der Variable `percentPlayed` der Wert des letzten Schmerzenslautes gespeichert.

Im Hauptprogramm muss noch beim Drücken der `ACTION`-Taste der Angriff ausgelöst werden, und die expliziten Aufrufe von `hero->update()` und `hero->draw()` entfernt werden, da der Held nun wie alle Sprites in der `characterList` der `World`-Klasse gespeichert wird. Auch wird nun das Spiel beendet, sobald der Held stirbt.

```
while (!key[KEY_ESC] && hero->isAlive()) {
    if (timerCounter) {
        joypad->poll();
```

```
if (joypad->button[Joypad::UP]) {
    hero->setDirection(Sprite::UP);
    hero->setMoving(true);
} else if (joypad->button[Joypad::DOWN]) {
    hero->setDirection(Sprite::DOWN);
    hero->setMoving(true);
} else if (joypad->button[Joypad::LEFT]) {
    hero->setDirection(Sprite::LEFT);
    hero->setMoving(true);
} else if (joypad->button[Joypad::RIGHT]) {
    hero->setDirection(Sprite::RIGHT);
    hero->setMoving(true);
} else if (joypad->button[Joypad::ACTION]) {
    hero->attack();
} else {
    hero->setMoving(false);
}

if (key[KEY_1]) {
    hero->setSpeed(0.1);
} else if (key[KEY_2]) {
    hero->setSpeed(0.12);
} else if (key[KEY_3]) {
    hero->setSpeed(0.14);
} else if (key[KEY_4]) {
    hero->setSpeed(0.16);
} else if (key[KEY_5]) {
    hero->setSpeed(0.18);
} else if (key[KEY_6]) {
    hero->setSpeed(0.2);
}

do {

    CharacterItor itor = world
                         ->getCharacters()
                           ->begin();
    CharacterItor end  = world
                         ->getCharacters()
                            ->end();
    for (;itor != end;) {
        Character* c = *itor;
        c->update();
```

```
                if (c->isAlive() || c == hero) {
                    ++itor;
                } else {
                    itor = world->getCharacters()
                                        ->erase(itor);
                    delete c;
                }
            }

            --timerCounter;
        } while (timerCounter >0);
        needsRedraw = true;
    }

    if (needsRedraw) {
        int x = MID(0,
                (hero->getX() - SCREEN_W/2),
                map->getWidth() * TILE_W - SCREEN_W);
        int y = MID(0,
                (hero->getY() - SCREEN_H/2),
                map->getHeight()* TILE_H - SCREEN_H);
        map->draw(x, y);

        CharacterItor itor = world->getCharacters()
                                        ->begin();
        CharacterItor end  = world->getCharacters()
                                        ->end();
        int count = 0;
        for (;itor != end; ++itor) {
            Character* c = *itor;
            c->draw(doubleBuffer, x, y);
            ++count;
        }

        needsRedraw = false;
        show();
    }
}
```

Damit das Ende nicht so abrupt kommt, wird der Bildschirm erst langsam mit schwarzen Linien gefüllt und dann das Game-Over-Bild angezeigt.

Abbildung 28.3: Der Game-Over-Screen

Das sieht alles schon ganz gut aus, das Problem ist nur, dass der Spieler keine Ahnung hat, wie es um die Gesundheit seines Helden gerade steht. Die Sprüche geben zwar eine grobe Orientierung, eine optisches Feedback wäre aber sicherlich besser.

Gesundheitsanzeige

Eine einfache, aber wirkungsvolle Methode den Gesundheitszustand darzustellen, ist es eine Bitmap mit einem Farbverlauf von Rot über Grün nach Gelb zu erstellen. Idealerweise sollte diese Bitmap 100 Pixel breit sein. Damit würde dann jeder Pixel einem Prozent entsprechen. Wenn man nun also weiß, dass der Spieler noch 50% Lebensenergie hat, dann bedeutet dies automatisch auch, dass nur 50 Pixel des Balkens angezeigt werden müssen.

Ist eine Anzeige einer Bitmap mit 100 Pixel nicht möglich, dann sollte eine Breite gewählt werden, die sich leicht auf die entsprechende Breite skalieren lässt. So ist der Farbverlauf in diesem Beispiel 200 Pixel breit.

Damit der Verlauf auch nicht auf der Karte untergeht, wird ein Rahmen um die Gesundheitsanzeige herum angezeigt.

Da es sich bei dem Verlauf und dem Rahmen um normale Bilddateien handelt, können Sie nun auch die Anzeige des Gesundheitsstatus beliebig verändern.

```
// Am Anfang der Datei
const int VITALITY_X = 10;
const int VITALITY_Y = 10;

// In der Main Funktion
BITMAP *gradient = load_bitmap("gradientred.tga",
                                                NULL);
BITMAP *border   = load_bitmap("border.tga", NULL);

int borderOfsX = (border->w - gradient->w) /2;
int borderOfsY = (border->h - gradient->h) /2;
```

Damit der Rahmen an der korrekten Position angezeigt wird, ist es wichtig die Breite des Rahmens zu kennen. Wenn wir davon ausgehen, dass der Rahmen symmetrisch ist, dann entspricht die Hälfte der Differenz der Bildgrößen von Rahmen und Farbverlauf der Rahmendicke.

Oder anders gesagt: Wenn man vom gesamten durch den Rahmen belegten Bereich die Größe des Bildes abzieht, dann bleibt nur noch der reine Rahmen übrig.

```
masked_blit(border, doubleBuffer,
            0, 0,
            VITALITY_X - borderOfsX,
            VITALITY_Y - borderOfsY,
            border->w, border->h);
if (hero->isAlive()) {
    blit(gradient, doubleBuffer,
         0, 0,
         VITALITY_X, VITALITY_Y,
         hero->getHealthStatus()*2, gradient->h);
}
```

Sie können den Rahmen auch ausgefallener entwerfen. Aber beachten Sie bitte, dass der Lebensenergiebalken immer zentriert über dem Rahmen angezeigt wird.

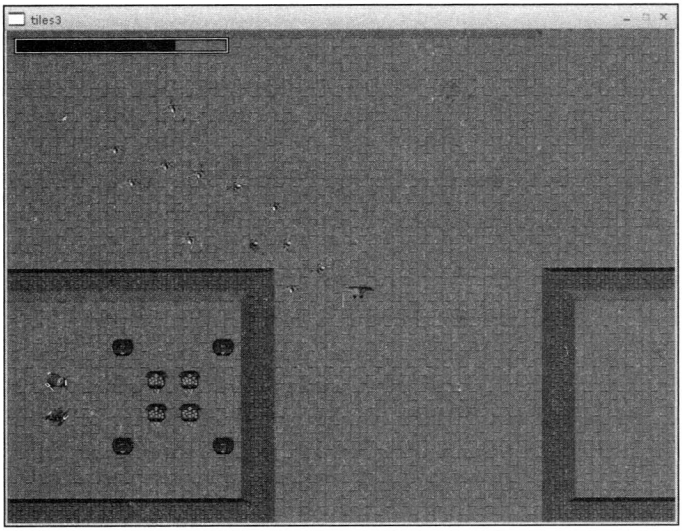


Abbildung 28.4: Die Gesundheitsanzeige

29 Magie und Spezialeffekte

In diesem Kapitel lernen Sie, wie Sie magische Geschosse erschaffen. Des Weiteren werden wir einige der Tiles in der Karte animieren und die restlichen Objekte implementieren. In einem letzten Schritt geben wir dem Level noch einen Ausgang – der natürlich zum nächsten Level führt.

Abrakadabra

Magische Effekte sind immer wieder gern gesehen in Rollenspielen. In Action-orientierten Spielen kann man mit ihnen große Mengen von Gegnern auf einmal besiegen, in Runden-basierten Spielen kann man durch Einsatz von Magie heilen, Gegner besiegen oder schwächen.

Mana

Der Einsatz von Zauberei benötigt in der Regel eine besondere Energiequelle, die als Mana bezeichnet wird. Benötigt ein Zauberspruch mehr Mana als ein Charakter hat, dann kann dieser Spruch nicht aktiviert werden.

Als Daumenregel gilt, dass mächtige Sprüche mehr Mana verbrauchen als schwache. Dadurch soll verhindert werden, dass es der Spieler zu leicht hat.

Mana ist ein Attribut wie jedes andere auch – und mit ein paar kleinen Änderungen an der `Character`-Klasse steht uns das Mana auch zur Verfügung. Die Anzahl der Attribute erhöht sich damit auf sieben.

```
enum {
    VITALITY = 0,
    STRENGTH = 1,
    ATTACK   = 2,
    DEFENSE  = 3,
    SPEED    = 4,
    IQ       = 5,
    MANA     = 6,
    ATTRIBUTE_COUNT = 7,
};
```

Der entsprechende Eintrag in der `game.ini` bekommt den überraschenden Namen »mana«:

```
void Character::load(const char* section) {
    const char *labels[] = {
        "vitality", "strength",
        "attack"  , "defense",
        "speed", "iq", "mana",
    };
    for (int a=0; a < ATTRIBUTE_COUNT; ++a) {
        attributes[a] = get_config_float(
                            section, labels[a], 1.0);
        maxAttributes[a] = attributes[a];
    }
    //....
}
```

Da der Manabalken ebenso wie der Lebensenergiebalken angezeigt werden soll, schreiben wir eine Methode, die analog zu der `getHealthStatus()`-Funktion der `PlayerCharacter`-Klasse arbeitet:

```
int PlayerCharacter::getManaStatus() {
    return (int) (100.0 *(attributes[MANA] /
                                maxAttributes[MANA]));
}
```

Nun muss der Balken nur noch angezeigt werden. Anstatt eines roten Balkens (wie für die Lebensenergie) nehmen wir einen blauen Balken.

```
const int MANA_X = 10;
const int MANA_Y = 40;

// ...

masked_blit(border, doubleBuffer,
            0, 0,
            VITALITY_X-borderOfsX,
            VITALITY_Y-borderOfsY,
            border->w, border->h);
if (hero->isAlive()) {
    blit(gradient, doubleBuffer,
         0, 0,
         VITALITY_X, VITALITY_Y,
         hero->getHealthStatus()*2, gradient->h);
}
masked_blit(border, doubleBuffer,
```

```
            0, 0,
            MANA_X-borderOfsX, MANA_Y-borderOfsY,
            border->w, border->h);
blit(manaGradient, doubleBuffer,
     0, 0, MANA_X, MANA_Y,
     hero->getManaStatus()*2, manaGradient->h);
```

Und schon erscheint unter dem Balken für die Lebensenergie der Balken für das Mana.

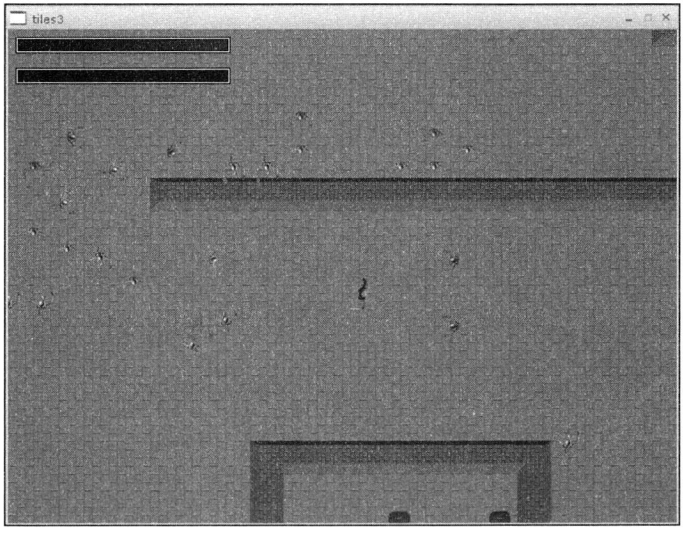


Abbildung 29.1: Mana-Anzeige

Jetzt haben wir das Mana – was nun noch fehlt, ist ein passender Zauberspruch.

Magische Geschosse

Als Beispiel-Zauberspruch nehme ich hier eine Reihe von magischen Geschossen, die ausgehend vom Spieler im näheren Umfeld einen gewaltigen Schaden unter den Monstern anrichten.

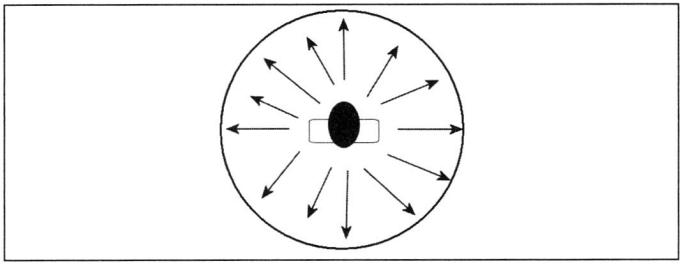


Abbildung 29.2: Verteilung der magischen Geschosse

Die Geschosse sollen sich möglichst gleichmäßig in alle Richtungen vom Spieler weg entfernen.

Um dies zu erreichen, wird für jedes Geschoss der Winkel berechnet, in dem es sich vom Spieler entfernt. Ausgehend vom Winkel ergeben sich dann die Geschwindigkeiten in X- und Y-Richtung.

Alle Geschosse teilen sich die gleiche Grafik, die einmal global in der `World`-Klasse geladen wird.

```
const float pi = 3.14159265358979323846264338327 95;

void World::magicAt(int x, int y) {
    if (bulletGFX == NULL) {
        bulletGFX = load_bitmap("bullet.tga", NULL);
    }
    for (int a=0; a < 48; ++a) {
        float angle = ((float) rnd((int) (200.0 * pi)))
                      / 100;
        float dx = 8 * cos(angle);
        float dy = 8 * sin(angle);

        Bullet *bullet = new Bullet(bulletGFX, 16, 16,
                                    dx, dy);
        bullet->setPosition(getHero()->getX(),
                            getHero()->getY());

        spriteList->push_back(bullet);
    }
}
```

Damit sich die Geschosse wirklich möglichst kreisförmig vom Spieler entfernen, ist eine höhere Genauigkeit bei der Position und den Geschwindigkeiten notwendig.

Die `Bullet`-Klasse leitet aus diesem Grund die entsprechenden Funktionen der `Sprite`-Klasse ab.

```
// bullet.h
#ifndef BULLET_HEADER
#define BULLET_HEADER

#include <allegro.h>
#include "sprite.h"

class Bullet: public Sprite {

private:
    float fdx, fdy;
    float fx, fy;

    int timeOut;

public:
    Bullet();
    Bullet(BITMAP *img, int fw, int fh,
                        float dx, float dy);


    virtual void update();
    virtual void draw(BITMAP *dest, int ox, int oy);

    virtual void setX(int x);
    virtual void setY(int y);
    virtual void setPosition(int x, int y);
};

#endif

// bullet.cpp
#include "bullet.h"
#include "world.h"
#include "character.h"


Bullet::Bullet(): Sprite() {
}
```

```
Bullet::Bullet(BITMAP *img, int fw, int fh, float dx, float dy) :
Sprite(img, fw, fh) {
    fdx = dx;
    fdy = dy;

    for (int a=0; a < countFrames; ++a) {
        deltaX[a] = deltaY[a] = 0;
    }
    moving = true;
    alive = true;
    timeOut = 30;

    if (ABS(dx) > ABS(dy)) {
        if (dx <0) {
            curDirection = LEFT;
        } else {
            curDirection = RIGHT;
        }
    } else {
        if (dy <0) {
            curDirection = UP;
        } else {
            curDirection = DOWN;
        }
    }
}


void Bullet::update() {
    updateCounter = 1.0;

    fx += fdx;
    fy += fdy;

    x = (int) fx;
    y = (int) fy;

    Sprite::update();
    if (lastCollision != NULL) {
        Character *c = (Character*) lastCollision;
        if (c != world->getHero()) {
            float vita =
                  c->getAttribute(Character::VITALITY);
            vita -= 10.0;
            if (vita <= 0.0) {
```

```
                vita = 0.0;
                c->setAlive(false);
            }
            c->setAttribute(Character::VITALITY, vita);
        } else {
            moving = true;
        }
    }
    --timeOut;
    if (!moving || timeOut <= 0) {
        setAlive(false);
    }
}

void Bullet::draw(BITMAP *dest, int ox, int oy) {
    masked_blit(frames, dest,
                curFrame * frameWidth,
                0,
                x-ox, y-oy,
                frameWidth, frameHeight);
}


void Bullet::setX(int x) {
    Sprite::setX(x);
    fx = x;
}
void Bullet::setY(int y) {
    Sprite::setY(y);
    fy = y;
}

void Bullet::setPosition(int x, int y) {
    Sprite::setPosition(x, y);
    fx = x;
    fy = y;
}
```

Der Constructor setzt je nach übergebenem Deltawert die Richtung (`curDirection`) des Sprites. Dies ist für die Kollisionsabfrage unbedingt notwendig. Wenn Sie also eigene Zaubersprüche implementieren, denken Sie unbedingt an das Setzen der aktuellen Richtung Ihres Sprites.

In der `update()`-Methode wird bei Kollision mit einem Monster die Lebensenergie des Monsters um zehn verringert.

Neu hinzugekommen ist die `timeOut`-Variable, welche die Lebensdauer des Geschosses angibt. Im Constructor wird dieser Wert auf 30 gesetzt, was dazu führt, dass jedes Geschoss nach einer halben Sekunde verschwindet – unabhängig davon, ob es ein Monster trifft oder nicht.

Ein Druck auf die `USE`-Taste (was dem 2ten Joypad Knopf bzw. der - Taste entspricht) ruft die `magic()`-Methode der `PlayerCharacter`-Klasse auf, die dann wiederum die `magicAt()`-Methode der `World`-Klasse aufruft.

```
joypad->poll();
if (joypad->button[Joypad::UP]) {
    hero->setDirection(Sprite::UP);
    hero->setMoving(true);
} else if (joypad->button[Joypad::DOWN]) {
    hero->setDirection(Sprite::DOWN);
    hero->setMoving(true);
} else if (joypad->button[Joypad::LEFT]) {
    hero->setDirection(Sprite::LEFT);
    hero->setMoving(true);
} else if (joypad->button[Joypad::RIGHT]) {
    hero->setDirection(Sprite::RIGHT);
    hero->setMoving(true);
} else if (joypad->button[Joypad::ACTION]) {
    hero->attack();
} else if (joypad->button[Joypad::USE]) {
    hero->magic();
} else {
    hero->setMoving(false);
}
```

In der `magic()`-Methode wird erst überprüft, ob der Spieler noch einen ausreichend hohen Manavorrat hat, um den Zauberspruch wirken zu können.

Reicht das Mana, wird ein Soundeffekt abgespielt, und die Geschosse werden auf die Reise geschickt. Ist nicht genug Mana vorhanden, dann tut dies der Held mittels Sprachausgabe kund.

In beiden Fällen wird jedoch ein Time-Out von einer Sekunde gesetzt, um den ununterbrochenen Einsatz von Zaubersprüchen zu verhindern.

```
void PlayerCharacter::magic() {
    if (magicTimeOut) {
        return;
    }
```

```
    float mana = getAttribute(MANA);
    if (mana >= 1.0) {
        playSound(samples[MAGIC_SND], 200);
        world->magicAt(x,y);
        mana -= 1.0;
        setAttribute(MANA, mana);
    } else {
        playSound(samples[MANA_SND], 200);
    }
    magicTimeOut = 60;
}
```

Das Ergebnis sieht dann in etwa so aus wie in Abbildung 29.3.

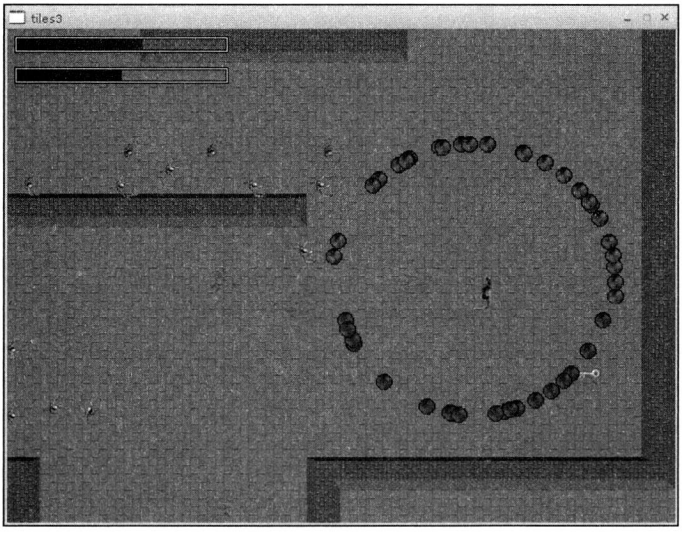


Abbildung 29.3: Zauberspruch in Aktion

Da `Bullet` von Sprite abgeleitet ist, erbt diese Klasse automatisch die Kollisionsabfrage. Zaubersprüche sind also deutlich weniger effektiv, wenn sie nah an Wänden gesprochen werden, da sich vermutlich immer einige Monster im Schutz der Mauern befinden.

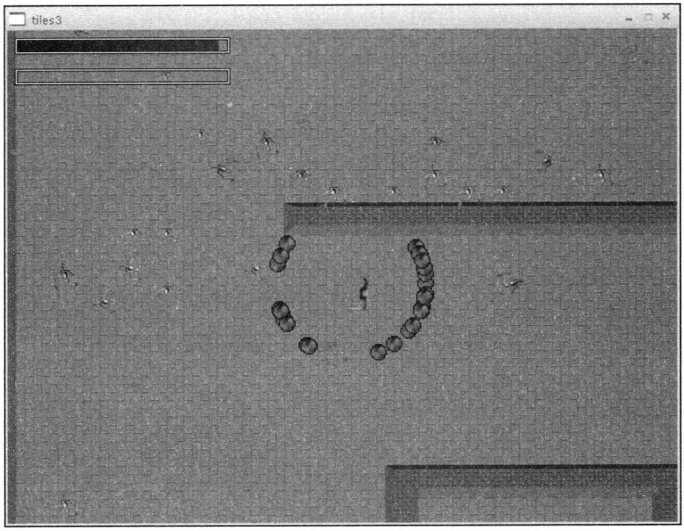


Abbildung 29.4: Mauern blockieren magische Geschosse

Wenn Sie den Manavorrat des Spielers in der `game.ini` auf zwei setzen, dann kann er nur zwei Zaubersprüche benutzen bevor sein Mana erschöpft ist. Und derzeit werden die Manatränke zwar aufgenommen, aber sie füllen den Vorrat noch nicht auf.

Magische Objekte

Die `objectTaken()`-Methode reagiert bisher noch nicht auf alle Objekttypen. Es wird Zeit, das zu ändern.

Die meisten magischen Gegenstände erhöhen entweder ein Attribut für kurze Zeit oder füllen ein Attribut auf.

Ein Geschwindigkeitstrank erhöht zum Beispiel die Geschwindigkeit des Spielers für fünf Sekunden auf das Vierfache, während Essen nur einen Teil des erlittenen Schadens wieder ausgleicht – aber niemals die maximale Lebensenergie erhöht.

```
bool PlayerCharacter::objectTaken(int id) {
    bool result = true;

    float maxVita = maxAttributes[VITALITY];
    float curVita = attributes[VITALITY];

    switch (id) {
        case World::DOOR:
        case World::LOCKED_CHEST:
            if (keyCount == 0) {
                result = false;
            } else {
                --keyCount;
            }
            break;

        case World::SPEED_POTION:
            setAttribute(SPEED,
                                getAttribute(SPEED) *4);
            timeOut[SPEED] = 5 * 60;
            break;

        case World::KEY:
            ++keyCount;
        break;


        case World::FOOD25:
            setAttribute(VITALITY,
                 MAX(maxVita, curVita + maxVita*0.25));
            break;
        case World::FOOD50:
            setAttribute(VITALITY,
                 MAX(maxVita, curVita + maxVita*0.50));
            break;
        case World::FOOD75:
            setAttribute(VITALITY,
                 MAX(maxVita, curVita + maxVita*0.75));
            break;

        case World::HEALTH_POTION:
            setAttribute(VITALITY, maxVita);
            break;
        case World::MANA_POTION:
            setAttribute(MANA, maxAttributes[MANA]);
```

```
            break;
        case World::SHIELD_POTION:
            setAttribute(DEFENSE,
                                getAttribute(DEFENSE) *3);
            timeOut[DEFENSE] = 5 * 60;
            break;
        case World::STRENGTH_POTION:
            setAttribute(STRENGTH,
                               getAttribute(STRENGTH) *2);
            timeOut[STRENGTH] = 5 * 60;
            break;
    }

    if (result) {
        score += world->getScore(id);

        playSound(samples[YEAH_SND], 250);
        curBubble = YEAH_SND;
        bubbleTimeOut = 60;
    }

    return result;
}
```

Die Gegenstände haben folgende Wirkung:

Name	Wirkung	Dauer
FOOD25	+25% Leben	sofort
FOOD50	+50% Leben	sofort
FOOD75	+75% Leben	sofort
HEALTH_POTION	100% Leben	sofort
MANA_POTION	100% Mana	sofort
SHIELD_POTION	3fache Verteidigung	5 Sekunden
STRENGTH_POTION	2fache Stärke	5 Sekunden
SPEED_POTION	4fache Geschwindigkeit	5 Sekunden

Tabelle 29.1: Wirkung der magischen Objekte

Durch das gezielte Platzieren von magischen Objekten können Sie Level interessanter gestalten. Wenn Sie zum Beispiel die Geschwindigkeitstränke so platzieren, dass man ungefähr fünf Sekunden braucht, um von einem Trank zum nächsten zu gelangen, dann regen Sie damit den Spieler an durch den Kerker zu rasen. Er wird dabei natürlich eine enorme Menge Monster anlocken. Wenn am Ende der Spur aus Geschwindigkeitstränken nun eine Sackgasse ist, dann dürfte es sich interessant gestalten, aus dieser Situation wieder herauszukommen.

Sie können dafür sorgen, dass es sich für den Spieler lohnt, das Risiko einzugehen, indem Sie jede Menge Schatztruhen am Ende des Weges platzieren. In Kombination mit Heil- und Manatränken könnte dies sogar dafür sorgen, dass der Spieler mit heiler Haut aus der Sache rauskommt.

Animierte Kacheln

Bisher waren die Kacheln statisch. Bisher. In wenigen Augenblicken werden wir die Bewegung in den Kerkerboden bringen.

Das von uns benützte Tile Set enthält drei Varianten des gleichen Pentagramms. Wenn diese nacheinander angezeigt werden, dann scheint das Pentagramm zu glühen.

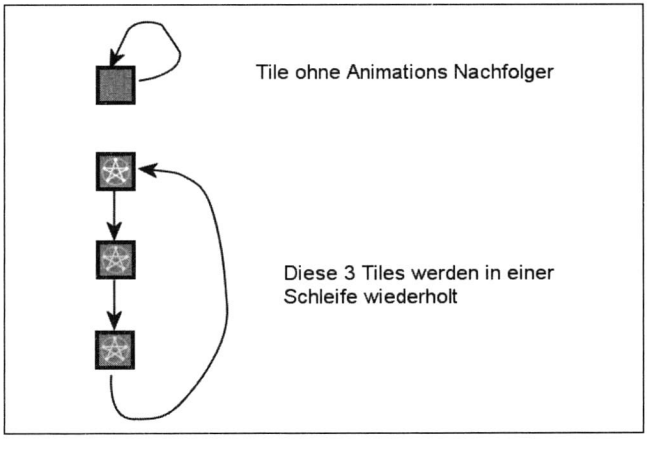


Abbildung 29.5: Tile Animation

Aber wie animieren wir die Tiles?

Die einfachste Lösung nenne ich »Eine Kachel sagts der anderen.« Das Prinzip ist sehr direkt: Für jede Kachel wird die Kachel gespeichert, die in der Animation auf sie folgt. Ist eine Kachel nicht Teil einer Animation, dann ist ihr eigener Index ihr Nachfolger.

Für diese Methode der Animation benötigen wir ein weiteres `int`-Array, das so viele Einträge hat wie Kacheln in der Tile-Grafik-Datei sind. Standardmäßig zeigt jeder Eintrag auf sich selbst:

```
void Map::setTiles(BITMAP *tiles, int tw, int th,
                   int firstWalkable) {
    this->tiles = tiles;
    tileWidth   = tw;
    tileHeight  = th;

    this->firstWalkable = firstWalkable;

    // Anim Array ggf. löschen
    if (anim) {
        delete[] anim;
    }
    int count = tiles->w / tw;

    // und in der korrekten grösse
    // neu erschaffen
    anim = new char[count];
    for (int a=0; a < count; a++) {
        // alle tiles zeigen auf sich selbst
        // keine Animation
        anim[a] = a;
    }
}
```

Die Animationssequenzen werden dann, genauso wie die Animationsgeschwindigkeit, in der `game.ini` festgelegt.

```
[tile.tga]
anim70=71
anim71=72
anim72=70

animDelay=20
```

Für jedes animierte Tile muss ein Eintrag mit dem jeweiligen Nachfolger vorhanden sein, damit die `readLevel()`-Methode der `World`-Klasse diese Information auslesen kann.

```
bool World::readLevel(int id) {
    // … Yadda - Yadda
    for (int a=0; a < tileCount; ++a) {
        tile2object[a] = NONE;
        sprintf(tileAnimName, "anim%i", a);
        map->setAnimTile( a,
            get_config_int(tileName, tileAnimName, a));
    }
    map->setAnimDelay(
            get_config_int(tileName, "animDelay", 10));
    // … Yadda - Yadda
}
```

Beim Überprüfen, ob ein Nachfolger gesetzt ist, wird der aktuelle Index als Defaultwert übergeben. Dadurch wird sicher gestellt, dass jedes Tile immer einen gültigen Nachfolger hat – selbst wenn keiner in der `game.ini` gesetzt wurde.

Jetzt sind alle Informationen, die wir zum Animieren der Kacheln benötigen, vorhanden. Damit sich wirklich etwas auf dem Schirm tut, muss allerdings noch die `draw()`-Methode der `Map`-Klasse von den animierten Tiles in Kenntnis gesetzt werden.

Der `animCounter` wird um eins erniedrigt. Erreicht er den Wert Null, dann wird es Zeit, das nächste Tile in der Animation anzuzeigen.

```
void Map::draw(int wx, int wy) {
    int pos = 0;

    wx = MID(0, wx, maxX);
    wy = MID(0, wy, maxY);

    int ox = wx % tileWidth;
    int oy = wy % tileHeight;
    wx /= tileWidth;
    wy /= tileHeight;


    animCounter--;
    if (animCounter <=0) {
        animCounter = animDelay;
        for (int y=0; y < visibleTilesY+1; ++y) {
            pos = wx * d + (y+wy) * lineSize;
            for (int x=0; x < visibleTilesX+1; ++x) {
                for (int z=0; z < d; ++z) {
                    if (pos < size) {
                        drawTile(dest,
                                x*tileWidth-ox,
```

```
                                y*tileHeight-oy,
                                data[pos]);
                        data[pos] = anim[data[pos]];
                    }
                    ++pos;
                }
            }
        }
    } else {
        for (int y=0; y < visibleTilesY+1; ++y) {
            pos = wx * d + (y+wy) * lineSize;
            for (int x=0; x < visibleTilesX+1; ++x) {
                for (int z=0; z < d; ++z) {
                    if (pos < size) {
                        drawTile(dest, x*tileWidth-ox,
                                       y*tileHeight-oy,
                                       data[pos]);
                    }
                    ++pos;
                }
            }
        }
    }
}
```

Der langen Mühe kurzer Lohn ist ein unheimlich blinkendes Pentagramm in unserem Kerker.

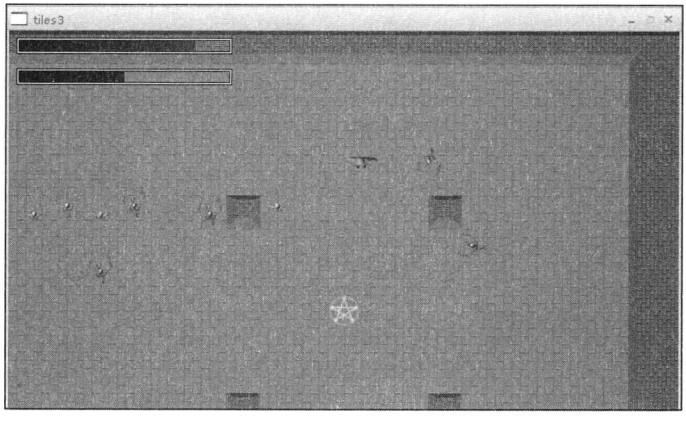


Abbildung 29.6: Das animierte Pentagramm

Level wechseln

Immer nur im gleichen Kerker herumzurennen ist auf Dauer doch eher langweilig. Aus diesem Grund fügen wir nun kurz entschlossen einen Ausgang hinzu – und zwar das soeben animierte Pentagramm.

Sobald sich der Spieler auf diese Kachel stellt soll er in den nächsten Level teleportiert werden. Falls es keinen weiteren Level mehr gibt, dann hat er gewonnen, und das Spiel ist zu Ende.

Die Vorgehensweise ist wie folgt:

- ✔ Wir definieren eine Position für den Ausgang (exitX, exitY). Betritt der Spieler den Bereich des Ausgangs, dann wird der nächste Level geladen.
- ✔ Gibt es keinen weiteren Level mehr, dann hat der Spieler gewonnen.

Dadurch wird es nun notwendig, das Hauptprogramm neu zu verschachteln. Zum einen wird der eigentliche Game Loop benötigt, darüber hinaus brauchen wir nun aber eine zweite Schleife, in der neue Level geladen werden können.

```
int main(int , char**) {
    init(640, 480, 60, false);

    joypad = new Joypad();
    world = new World();
    int curLevel = 0;
    world->readLevel(curLevel);


    PlayerCharacter *hero  = world->getHero();
    Map *map = world->getMap();
    map->setTargetBitmap(doubleBuffer);
    bool needsRedraw = false;


    BITMAP *gradient     = load_bitmap(
                               "gradientred.tga", NULL);
    BITMAP *manaGradient = load_bitmap(
                               "gradientblue.tga", NULL);
    BITMAP *border       = load_bitmap(
                               "border.tga", NULL);
```

```
int borderOfsX = (border->w - gradient->w) /2;
int borderOfsY = (border->h - gradient->h) /2;

timerCounter = 0;
syncTimer(&timerCounter);
bool gameWon = false;
while (!key[KEY_ESC]
       && hero->isAlive() &&!gameWon) {

    while (!key[KEY_ESC]
            && hero->isAlive()
            && !world->isExitFound()) {

        if (timerCounter) {
            joypad->poll();
            if (joypad->button[Joypad::UP]) {
                hero->setDirection(Sprite::UP);
                hero->setMoving(true);
            } else if (joypad->button
                                [Joypad::DOWN]) {
                hero->setDirection(Sprite::DOWN);
                hero->setMoving(true);
            } else if (joypad->button
                                [Joypad::LEFT]) {
                hero->setDirection(Sprite::LEFT);
                hero->setMoving(true);
            } else if (joypad->button
                                [Joypad::RIGHT]) {
                hero->setDirection(Sprite::RIGHT);
                hero->setMoving(true);
            } else if (joypad->button[Joypad::ACTION]) {
                hero->attack();
            } else if (joypad->button[Joypad::USE]) {
                hero->magic();
            } else {
                hero->setMoving(false);
            }

            if (key[KEY_1]) {
                hero->setSpeed(0.1);
            } else if (key[KEY_2]) {
                hero->setSpeed(0.12);
            } else if (key[KEY_3]) {
                hero->setSpeed(0.14);
            } else if (key[KEY_4]) {
                hero->setSpeed(0.16);
```

```
    } else if (key[KEY_5]) {
        hero->setSpeed(0.18);
    } else if (key[KEY_6]) {
        hero->setSpeed(0.2);
    }


    do {

        CharacterItor itor = world->
                  getCharacters()->begin();
        CharacterItor end  = world->
                   getCharacters()->end();
        for (;itor != end;) {
            Character* c = *itor;
            c->update();

            if (c->isAlive() || c == hero) {
                ++itor;
            } else {
                itor = world->
                             getCharacters()
                              ->erase(itor);
                delete c;
            }
        }
        SpriteItor sprItor = world->
                  getSpriteList()->begin();
        SpriteItor sprEnd  = world->
                  getSpriteList()->end();
        for (;sprItor != sprEnd;) {
            Sprite *s = *sprItor;
            s->update();
            if (s->isAlive()) {
                ++sprItor;
            } else {
                sprItor = world->
                         getSpriteList()
                          ->erase(sprItor);
                delete s;
            }
        }
        --timerCounter;
    } while (timerCounter >0);
    needsRedraw = true;
}
```

```
needsRedraw = true;

if (needsRedraw) {
    int x = MID(0,
            (hero->getX() - SCREEN_W/2),
             map->getWidth()
              * TILE_W - SCREEN_W);

    int y = MID(0,
             (hero->getY() - SCREEN_H/2),
             map->getHeight()
               * TILE_H - SCREEN_H);
    map->draw(x, y);

    CharacterItor itor = world->
                  getCharacters()->begin();
    CharacterItor end  = world->
                  getCharacters()->end();
    int count = 0;
    for (;itor != end; ++itor) {
        Character* c = *itor;
        c->draw(doubleBuffer, x, y);
        ++count;
    }
    SpriteItor sprItor = world->
                  getSpriteList()->begin();
    SpriteItor sprEnd  = world->
                  getSpriteList()->end();
    for (;sprItor != sprEnd; ++sprItor) {
        Sprite *s = *sprItor;
        s->draw(doubleBuffer, x, y);
    }
    masked_blit(border, doubleBuffer,
                0, 0,
                VITALITY_X-borderOfsX,
                VITALITY_Y-borderOfsY,
                border->w, border->h);
    if (hero->isAlive()) {
        blit(gradient, doubleBuffer, 0, 0,
             VITALITY_X, VITALITY_Y,
             hero->getHealthStatus()*2,
             gradient->h);
    }
    masked_blit(border, doubleBuffer, 0, 0,
                MANA_X-borderOfsX,
                MANA_Y-borderOfsY,
```

```
                                    border->w, border->h);
                        blit(manaGradient, doubleBuffer, 0, 0,
                             MANA_X, MANA_Y,
                             hero->getManaStatus()*2,
                             manaGradient->h);

                        needsRedraw = false;
                        show();
                    }
                }

                if (hero->isAlive() && world->isExitFound()) {
                    curLevel++;
                    if (world->readLevel(curLevel)) {
                        map = world->getMap();
                        map->setTargetBitmap(doubleBuffer);
                    } else {
                        gameWon = true;
                    }
                }

            }

            BITMAP *image = NULL;

            if (gameWon) {
                image = load_bitmap("won.tga", NULL);
            } else {
                image = load_bitmap("gameover.tga", NULL);
            }
            for (int a=0; a < SCREEN_H/2; ++a) {
                hline(doubleBuffer,0,a,SCREEN_W, 0);
                hline(doubleBuffer,0,SCREEN_H-a,SCREEN_W, 0);
                show();
            }
            blit(image, doubleBuffer, 0, 0, 0, 0,
                 image->w, image->h);
            show();

            clear_keybuf();
            readkey();
            destroy_bitmap(image);
            destroy_bitmap(gradient);
            destroy_bitmap(border);

            return 0;
        } END_OF_MAIN()
```

Schafft der Spieler den Level, bekommt er anstatt der »Game Over«- eine »Well Done«-Meldung. Sie können hier auch eine Video-Sequenz abspielen und zur Bestenliste verzweigen.

Die Position des Ausgangs kann beim Laden des Levels aus der Position des Tiles hergeleitet werden, das den Ausgang markiert – wenn jemand nun noch so nett wäre, der `World`-Klasse zu sagen, welches Tile der Ausgang ist?

»I need an Exit!«

Wir können den Ausgang vom Prinzip so behandeln wie die Objekte – mit einer Ausnahme: Das Ausgangstile befindet sich nicht auf der Objektebene. Beim Durchsuchen der Karte müssen wir nun also eine zusätzliche Abfrage machen.

```
for (int y=0; y < map->getHeight(); ++y) {
    for (int x=0; x < map->getWidth(); ++x) {
        int tile = map->getTileAt(x,y,objectLayer);
        for (int a=0; a < monsterCount; ++a) {
            if (tile2monster[a] == tile) {
                map->setTileAt(x,y,objectLayer, 0);
                Monster *monster = new Monster(*
                                (monsterRepository[a]));
                monster->setPosition(x * tileWidth,
                                     y * tileHeight);
                characterList->push_back(monster);
            }
        }
        if (map->getTileAt(x,y,0) == exitTile) {
            exitX = x * tileWidth;
            exitY = y * tileHeight;
        }
    }
}
```

Jetzt muss nur noch die `PlayerCharacter`-Klasse überprüfen, ob wir uns auf einem Ausgang befinden und einem Levelwechsel steht nichts mehr im Wege.

```
void PlayerCharacter::update() {
    Character::update();
    // Yadda – yadda - yadda
```

```
    if (!(x + frameWidth/2 <= world->getExitX() ||
        x >= world->getExitX() + map->getTileWidth() ||
        y + frameHeight/2 <= world->getExitY() ||
        y >= world->getExitY() + world->getExitY()
                                + map->getTileHeight())) {
            world->signalExitFound();
    }
    // Yadda - yadda - yadda
}
```

Wenn sich der Spieler nun auf ein Pentagramm stellt, wird er in den nächsten Level teleportiert.

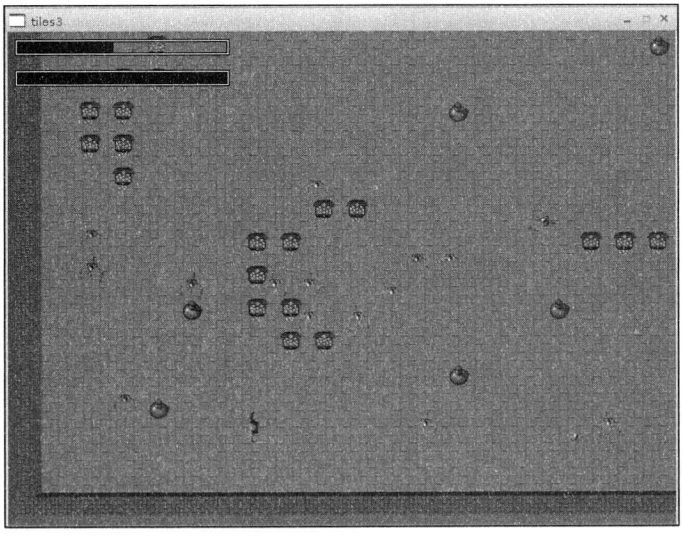


Abbildung 29.7: Neue Aufgaben warten auf den Helden

Der 2te Level hat ein sehr einfaches Layout, ist aber mit Schätzen und Monstern gefüllt. In der `game.ini` können die Einträge des ersten Levels für den zweiten wiederverwendet werden – allerdings muss natürlich der Name der Leveldatei angepasst werden.

```
[level1]
map=level1.map
tiles=tile.tga
objectLayer=1
```

Schafft es der Held auch diesmal sich zum Ende des Levels vorzukämpfen, dann wird er mit der Endgrafik belohnt.

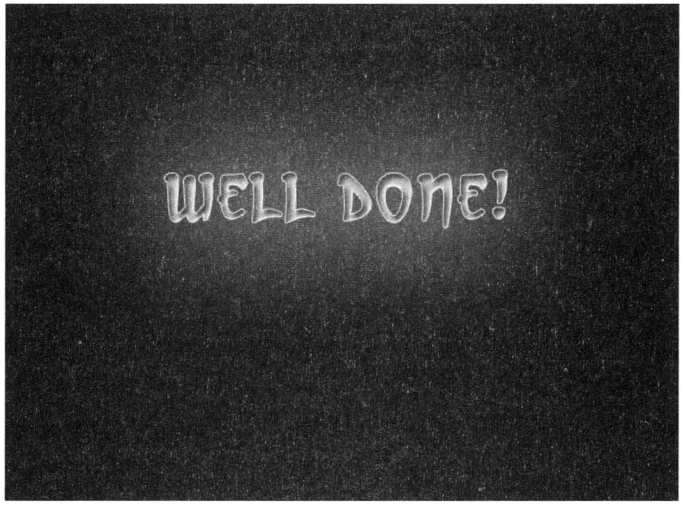


Abbildung 29.8: Das Gute hat gewonnen!

30 Unterschiedliche Charaktere und Levelanstieg

Eines der Kernelemente eines Rollenspiels ist die Verbesserung der Fertigkeiten. In diesem Kapitel lernen Sie, wie Sie unterschiedliche Charaktere entwerfen und deren Levelanstieg planen.

Krieger, Magier und Bogenschütze

Es gibt drei Archetypen von Charakteren in Rollenspielen. Dabei handelt es sich um den mächtigen Krieger, der sich auf seine Kraft verlässt, den Magier, der seine Gegner durch Magie bezwingt und den Bogenschützen, der schnell und gewandt ist und die Gegner lieber aus sicherer Entfernung besiegt.

Der Krieger

Ein Krieger ist stark, zäh und zieht seine Waffe dem Hokus-Pokus vor. Mit anderen Worten, er hat hohe Werte in Kraft und Vitalität, allerdings ist sein Manavorrat sehr begrenzt.

Überträgt man dies in die `game.ini`, dann sieht das so aus:

```
[hero]
vitality=100
strength=10
attack=1
defense=1
speed=0.2
mana=2.0
```

Wenn Ihnen das bekannt vorkommt, dann liegt das daran, dass dies die aktuellen Werte aus der `game.ini` unseres Helden sind.

Wenn der Krieger einen Level aufsteigt, dann steigen seine Vitalität und seine Stärke am schnellsten an. Sie könnten den Maximalwert um 20 für Vitalität und die Stärke um 5 Punkte steigern.

Der Angriffswert könnte um 1 Punkt steigen, die Verteidigung um 0.5 Punkte. Die wenigsten Krieger sind für ihre Geschwindigkeit bekannt – diese steigt nur um 0.025 Punkte pro Level an.

Abbildung 30.1: Der Barbar/Krieger

Der Mana-Wert steigt nicht an, sondern bleibt immer auf dem Ausgangwert.

Level	Vita	Stärke	Angriff	Vert.	Gesch.	Mana
1	100	10	1	1	0.2	2
2	120	15	2	1.5	0.225	2
3	140	20	3	2	0.25	2
4	160	25	4	2.5	0.275	2
5	180	30	5	3	0.3	2

Level	Vita	Stärke	Angriff	Vert.	Gesch.	Mana
6	200	35	6	3.5	0.325	2
7	220	40	7	4	0.350	2
8	240	45	8	4.5	0.375	2
9	260	50	9	5	0.4	2
10	280	55	10	5.5	0.425	2

Tabelle 30.1: Entwicklung des Kriegers

Der Magier

Der Magier ist das Gegenstück zum Krieger. Wo der Krieger auf seine Muskeln setzt, da setzt der Zauberer auf Magie. Körperlich ist der Magier am schwächsten – weder seine Stärke noch die Gesundheit sind wirklich berauschend. Er ist etwas schneller als der Krieger. Dies ist auch notwendig, da er in einem Nahkampf kaum eine Chance hat. Im Nahkampf sind seine Angriffe nicht sehr beeindruckend. Steht aber kein Gegner vor ihm, dann schießen Feuerbälle aus seinen Händen, die bei den Gegnern einen ziemlichen Eindruck hinterlassen.

Abbildung 30.2: Der Zauberer

Zauberer brauchen einen hohen Manavorrat, da ihr normaler Angriff (der Feuerball) ebenfalls Zauberkraft verbraucht. Um seine körperliche Schwäche auszugleichen, benutzt er einen anderen Zauberspruch als der Krieger. Anstatt einer Explosion von magischen Geschossen besitzt der Magier einen Vitalitätszauber. Dieser wirkt wie eine Mahlzeit (Vitalität + 25% des Maximums), zusätzlich wird die Geschwindigkeit für fünf Sekunden um 0.1 erhöht.

```
[wizard]
vitality=60
strength=5
attack=0.5
defense=0.5
speed=0.3
mana=4.0
```

Bei einem Levelaufstieg geht der Manawert am deutlichsten nach oben. Für jede Stufe bekommt der Magier 3 weitere Mana-Punkte.

Die Vitalität steigt um 10, Stärke um 2, Angriff und Verteidigung um jeweils 0.5 Punkte. Die Geschwindigkeit des Magiers steigt um 0.02 pro Stufe.

Level	Vita	Stärke	Angriff	Vert.	Gesch.	Mana
1	60	5	0.5	0.5	0.3	5
2	70	7	1	1	0.32	8
3	80	9	1.5	1.5	0.34	11
4	90	11	2	2	0.36	14
5	100	13	2.5	2.5	0.38	17
6	110	15	3	3	0.4	20
7	120	17	3.5	3.5	0.350	23
8	130	19	4	4	0.375	26
9	140	21	4.5	4.5	0.4	29
10	150	23	5	5	0.42	32

Tabelle 30.2: Entwicklung des Magiers

Bitte beachten Sie, dass der Magier im Nahkampf dem Krieger wirklich deutlich unterlegen ist. Auf der ersten Stufe kann der Zauberer nur etwa 3 Treffer einstecken, bevor er das Zeitliche segnet. Durch seine geringe Verteidigung steigt die Wirksamkeit der gegnerischen Angriffe, und seine geringe Lebensenergie tut ein Übriges. Der Vitalitätszauber wird da schnell zu einem Lebensretter.

Der Bogenschütze

Der Bogenschütze ist schnell. Wirklich schnell. Er verschießt seine Pfeile mit tödlicher Präzision, ist aber im Nahkampf nicht sonderlich gut – wenn auch besser als der Magier.

Die Pfeile des Bogenschützen richten nicht so viel Schaden an wie die Feuerbälle des Magiers, allerdings verbrauchen sie auch keine Zauberkraft.

```
[archer]
vitality=80
strength=8
attack=1
defense=1
speed=0.3
mana=2.0
```

Der Zauberspruch des Bogenschützen friert alle sichtbaren Gegner mit einer Chance von 50% für 5 Sekunden ein. Im Idealfall kann es also sein, dass alle Gegner in Reichweite für 5 Sekunden nichts machen können – im schlechtesten Fall sind alle Gegner unbeeindruckt. Eine Alternative wäre es, jeden Gegner einzufrieren, aber die Zeitdauer zufällig zu wählen. So könnten einige Monster nach 0.5 Sekunden wieder aufwachen, andere erst nach 5 Sekunden.

Bei einem Levelaufstieg steigen Geschwindigkeit, Vitalität und Verteidigung am deutlichsten an. Die restlichen Werte steigen eher verhalten.

Level	Vita	Stärke	Angriff	Vert.	Gesch.	Mana
1	80	8	1	1	0.3	2
2	95	11	1.55	2	0.33	3
3	110	14	2.1	3	0.39	4

Level	Vita	Stärke	Angriff	Vert.	Gesch.	Mana
4	125	17	2.65	4	0.42	5
5	140	20	3.2	5	0.45	6
6	155	23	3.75	6	0.48	7
7	170	26	4,3	7	0.51	8
8	185	29	4.85	8	0.54	9
9	200	32	5,4	9	0.58	10
10	215	35	5.95	10	0.61	11

Tabelle 30.3: Entwicklung des Bogenschützen

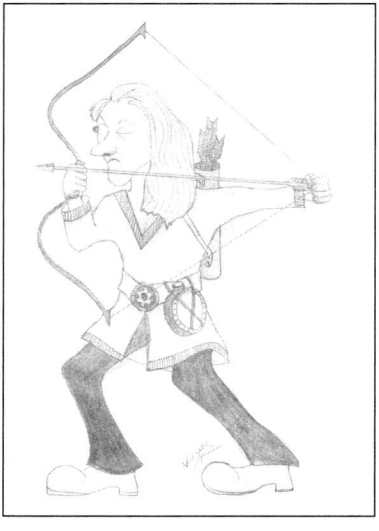


Abbildung 30.3: Der Bogenschütze

Weitere Klassen

Sie können natürlich auch weitere Klassen zu Ihrem Spiel hinzufügen. Die drei bisher beschriebenen sind nur die grundlegenden Archetypen. Ich stelle Ihnen hier noch ein paar weitere mögliche Klassen vor – eventuell ist ja eine dabei, die Ihnen zusagt?

Der Panzer

Der Panzer verlässt sich auf seine Verteidigung und seinen hohen Vitalitätswert. Wann immer irgendwo eine größere Menge von Monstern ist, der Panzer ist genau in der Mitte von ihnen. Ein möglicher Zauberspruch wäre eine kleine Explosion von magischen Geschossen, die nur Gegner im näheren Umfeld beeinflusst.

Ein Ritter, der sich auf seine Rüstung verlässt, wäre ein typischer Panzer.

Der Blutmagier

Der Blutmagier benutzt seine eigene Lebensenergie für seine Zaubersprüche. Er hat kein Mana, sondern verbraucht für jeden Zauberspruch 5 Punkte Vitalität. Von der gesundheitlichen Entwicklung ist er etwa so wie der Bogenschütze, seine magischen Geschosse sind nicht so mächtig wie die eines normalen Zauberers, allerdings ist er dafür im Nahkampf besser bewandert. Manatränke bewirken eine 50% Heilung.

Der Vampir

Nicht so stark wie der Krieger, nicht so schnell wie der Bogenschütze aber dennoch nicht hilflos – der Vampir erhält für jeden erfolgreichen Treffer 2 Punkte Lebensenergie. Sein Zauberspruch hat eine 50% Chance den Gegner zu eliminieren (in diesem Fall bekommt er 5 Punkte Vitalität). Wird der Gegner nicht direkt besiegt, so ist er für 2 Sekunden gelähmt. Der Mana Vorrat des Vampirs ist auf 2 Punkte begrenzt, und steigt auch nicht an. Aber immer wenn der Vampir bei voller Gesundheit ist, füllt sich das Mana mit 0.1 Punkten pro Sekunde automatisch.

Der Priester

Der Priester besitzt die göttliche Gabe des Heilens. Sein Zauber bewirkt eine 50%ige Heilung. Von den restlichen Werten ist der Priester ein nicht

ganz so guter Kämpfer. Auf Stufe 10 sollten seine Werte in etwa denen eines Kriegers der 7ten oder 8ten Stufe entsprechen. Mit einer Ausnahme: der Mana Wert steigt jede Stufe um 0.2 Punkte an, ist also deutlich höher als der des Kriegers.

Wenn Sie möchten, dann können Sie dem Priester auch eine besondere Fertigkeit gegen untote Monster geben. So könnte er einen Angriffsbonus von 0.5 gegen Untote besitzen, oder seine Klassenstufe zu dem Schaden gegen diese Monster hinzuzählen können.

Der Berserker

Der Berserker hat die Fähigkeit sich in einen Kampfrausch zu versetzen. In diesem Kampfrausch hat er die doppelte Stärke, dreifache Vitalität, den dreifachen Angriffswert und den halben Verteidigungswert wie normal. Ein Kampfrausch dauert 5 Sekunden. Nach dieser Zeit sinken die Werte wieder. Hatte der Berserker am Ende des Rausches weniger als 3 Punkte Vitalität, dann stirbt er im Anschluss an den Kampfrausch.

Der Gestaltwandler

Ein Gestaltwandler ist die meiste Zeit ein Durchschnittstyp, kann sich jedoch durch Magie in ein Wesen mit anderen Eigenschaften verwandeln. Beispiele für Klassen sind Wertiere (Werwöfe, Wertiger, Werfledermäuse) oder Druiden beziehungsweise Schamanen, die eine Tiergestalt annehmen können.

In der Regel erfolgt die Verwandlung zu einem Extrem:

- ✔ Verwandlung in eine starke, aber langsame Kreatur (z.B. ein Bär).
- ✔ Verwandlung in eine schnelle, aber nicht so starke Kreatur (z.B. Wolf).

Der Schurke

Der Schurke ist dem Bogenschützen sehr ähnlich. Allerdings ist er nicht ganz so schnell, hat dafür aber einen höheren Angriffswert. Ein Schurke kann verschlossene Schatztruhen öffnen, ohne einen Schlüssel zu benutzen. Darüber hinaus hat er die Fähigkeit unsichtbar zu werden. Aktiviert er seinen Zauber, dann sehen ihn die Monster für 5 Sekunden nicht mehr, und er kann sich an ihnen vorbeischleichen.

Alternatives Verbesserungssystem

Die bisher beschriebenen Klassen spielen sich alle von Anfang an recht unterschiedlich. Der Bogenschütze flitzt durch die Gegend, der Magier beharkt seine Gegner aus sicherem Abstand und der Krieger wirft sich mitten in das Getümmel.

Der Vorteil dieser Methode ist, dass der Spieler sofort die Unterschiede zwischen den einzelnen Klassen erkennt. Der Nachteil ist, dass Sie sicher stellen müssen, dass alle Level wirklich mit allen Charakteren spielbar sind.

Und gerade auf den unteren Stufen ist die Gefahr hoch, dass Klassen wie Magier und Bogenschützen von zu vielen Gegnern einfach überrannt werden.

Sie können dieses Problem umgehen oder zumindest abschwächen, wenn Sie jedem Charakter die gleichen Ausgangswerte geben. Dann haben alle Charaktere die gleichen Chancen auf den unteren Leveln. Sobald Sie dann Klassenstufen hinzugewinnen, ändert sich auch das Spielgefühl mit diesen Charakteren, da sie sich in verschiedene Richtungen entwickeln.

Der Vorteil ist zum einen, dass der Spieler langsam in die Eigenheiten einer Klasse eingeweiht wird, und dass es für Sie einfacher ist die Level zu entwerfen.

Der Nachteil ist jedoch, dass sich alle Klassen anfangs gleich spielen. Sie müssen sicher stellen, dass dem Spieler bewusst ist, dass die Charaktere sich jedoch im Laufe des Spieles in andere Richtungen entwickeln – sonst könnte es Ihnen passieren, dass er alle Klassen für größtenteils gleich hält und sich gar nicht erst die Mühe macht, mit allen Charakteren zu spielen.

Erfahrungspunkte und Stufengrenzen

Normalerweise bekommt ein Spieler für jedes besiegte Monster eine bestimmte Anzahl von Erfahrungspunkten. Übersteigt die Summe der angesammelten Erfahrungspunkten eine Grenze, dann erreicht der Charakter die nächste Stufe.

Jede Stufe ist schwieriger zu erreichen als die vorherige. Eine einfache Methode dies zu erreichen, ist die Anzahl der nötigen Punkte durchgehend zu verdoppeln:

Stufe	Punkte
1	0
2	100
3	200
3	400
4	800
5	1600
6	3200
7	6400
8	12800
9	256000
10	512000

Tabelle 30.4: Erfahrungspunkte pro Stufe

Gibt ein Skelett 5 Punkte, dann braucht es 20 Skelette, um die 2 Stufe zu erreichen – aber 40, um von der 2ten auf die 3te Stufe zu kommen. Allerdings sind diese 40 Skelette einfacher zu besiegen, da ja der Charakter nun besser geworden ist. Um zu verhindern, dass ein Spieler versucht seinen Charakter mit leichten Monstern auf einen zu hohen Level zu bringen, kann man die Anzahl der vergebenen Erfahrungspunkte abhängig vom Stufenunterschied machen.

Spielerstufe – Monsterstufe	Erfahrungspunkte
-5	0
-4	1
-3	1
-2	2

Spielerstufe – Monsterstufe	Erfahrungspunkte
-1	3
0	5
+1	6
+2	8
+3	10
+4	12
+5	15

Tabelle 30.5: Zu vergebende Erfahrungspunkte

Ist der Spieler auf der gleichen Stufe wie das Monster, dann erhält er 5 Punkte. Ist das Monster aber auch nur eine Stufe unter ihm, bekommt er nur noch 3 Punkte. Ist das Monster 5 oder mehr Stufen unter ihm, dann bekommt er keinerlei Erfahrungspunkte mehr für den Sieg.

Ist das Monster stärker als der Spieler, dann bekommt er bis zu 10 Bonuspunkte für den Stufenunterschied. Es lohnt sich also die schwierigeren Monster anzugehen – allerdings geht man dann auch ein größeres Wagnis ein.

TEIL IV

Weiterführende Methoden

In diesem Teil gehen wir auf komplexere Themen ein: Spezialeffekte, Wegfindung, isometrische Grafik und Skriptsprachen werden behandelt.

31 Isometrische Karten

In diesem Kapitel behandeln wir die Grundlagen von Karten, die in einer isometrischen Ansicht dargestellt werden. Wir haben die Isometrieperspektive im Kapitel über Perspektiven in Spielen bereits kurz vorgestellt. In diesem Kapitel geht es um die Umsetzung dieser Perspektive in einem Spiel.

Grundlagen der Isometrie

Bei der Isometrie behalten alle Objekte ihre Größe, unabhängig von der Position im Raum. Es gibt also keine durch die Perspektive bedingten Verkleinerungen. Im einfachsten Fall können Sie sich ein isometrisches Spielfeld wie eine um 45 Grad gedrehte und leicht nach hinten gekippte normale Karte vorstellen.

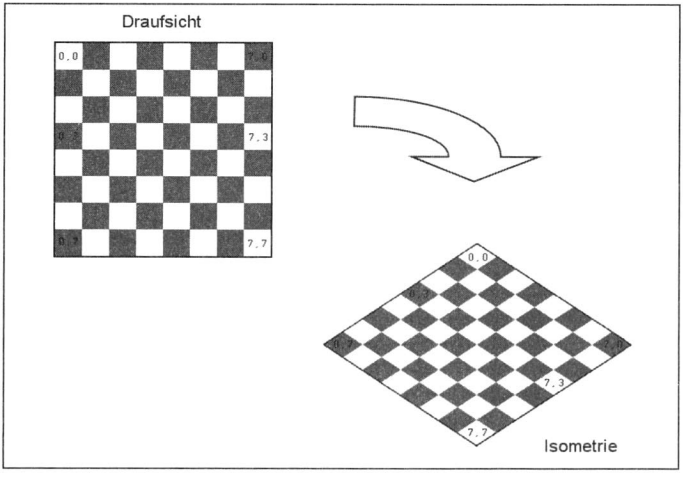


Abbildung 31.1: Draufsicht zu Isometrie

Wenn ein Sprite in Spielkoordinaten nach rechts geht, dann bewegt es sich auf dem Schirm diagonal nach rechts unten.

Bewegt sich das Spiel in der Welt diagonal, dann bewegt es sich auf dem Schirm entlang einer der Bildschirmachsen.

Um zum Beispiel von Feld (0,0) nach Feld (1,0) zu gelangen (ein Schritt nach rechts in den Koordinaten der Karte), muss sich das Sprite auf dem Bildschirm eine halbe Kachel nach rechts und eine halbe Kachel nach unten bewegen.

Bewegt sich das Sprite jedoch entlang der X-Achse (in Bildschirmkoordinaten), so verändert sich sowohl seine X- als auch die Y-Position.

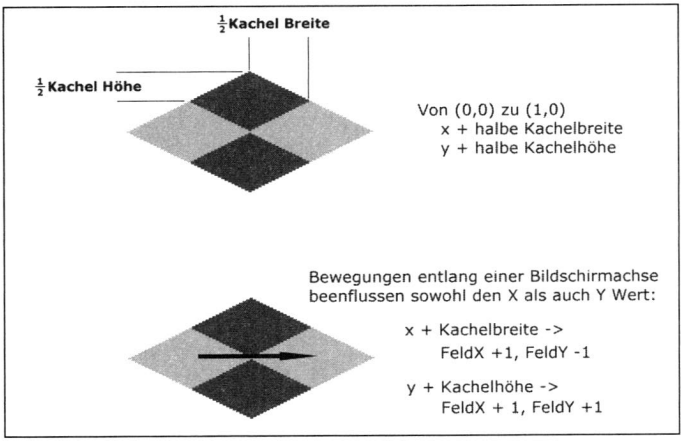


Abbildung 31.2: Bewegung im Koordinatensystem

Diese Art der Bewegung kann sehr verwirrend sein, erlaubt aber eine Bewegung im 3-dimensionalen Raum, ohne komplexe 3D-Berechnungen durchführen zu müssen.

Eine kurze Geschichte der Isometrie

Die ersten Spiele, die dies ausnutzten, waren »Q*Bert« von der Firma Gottlieb und »Zaxxon« von der Firma Sega, die beide im Jahre 1982 auf den Markt kamen (siehe Abbildung 31.3).

Im Jahre 1986 kam dann »Batman« von der Firma Ocean auf den Markt, ein Jahr später das Kultspiel »Head over Heels«, ebenfalls von Ocean und mit einer nahezu identischen Präsentation wie »Batman«.

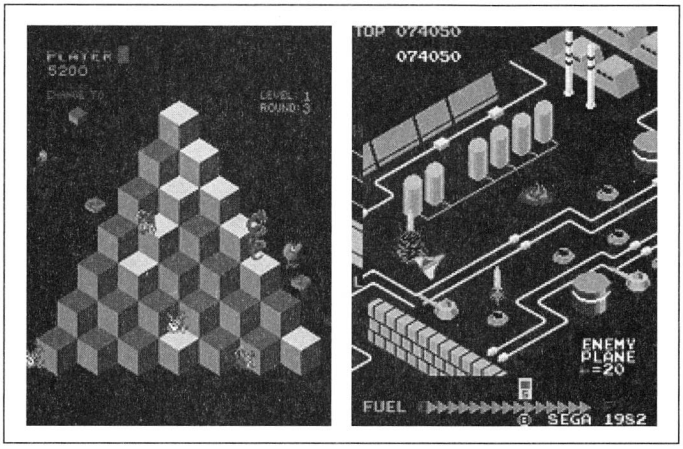


Abbildung 31.3: Q*Bert und Zaxxon

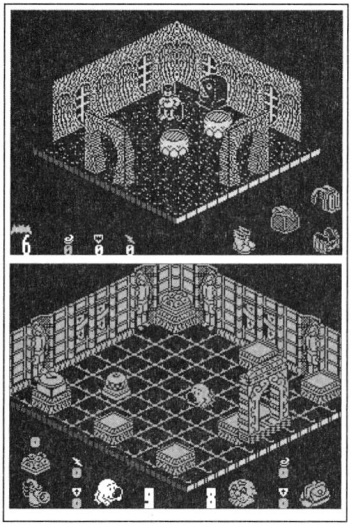


Abbildung 31.4: Batman und Head over Heels

Auf dem PC kam der große Durchbruch erst mit dem Spiel »UFO: Enemy Unknown« von der Firma Mythos Games aus dem Jahre 1994. Dieses Spiel war der Beginn einer ganzen Reihe von isometrischen Spielen, deren Höhepunkte sicherlich in Spielen wie »Diablo II« (Blizzard Entertainment), »Baldur's Gate II« (Bioware) und »Age of Empires« (Microsoft) liegen.

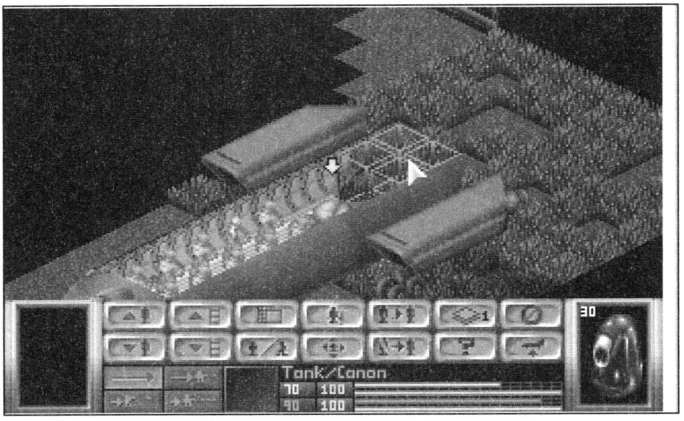


Abbildung 31.5: UFO: Enemy Unknown

Die Isometrie bietet einen guten Kompromiss von Realitätsnähe und Programmieraufwand. Womit wir wieder beim Thema sind.

Implementierung

Wenn Sie vom Feld (0, 0) ein Feld nach rechts auf das Feld (0, 1) gehen, dann bewegen Sie sich dabei um eine halbe Kachelbreite nach rechts, und eine halbe Kachelhöhe nach unten.

Gehen wir ein Feld nach unten, also von (0, 0) nach (0, 1), dann bewegen wir uns eine halbe Kachelbreite nach links und eine halbe Kachelhöhe nach unten.

Und genau auf diese Weise zeichnen wir auch eine isometrische Karte.

Ausgehend vom Ursprung, also dem Feld (0,0) berechnen wir die Startposition der Reiche von Tiles. Da je nach der Art und Weise wie die Gra-

fiken gezeichnet wurden nicht immer genau die Hälfte des Tiles als Offset genommen werden kann, haben wir Schrittweite als Konstanten definiert.

```
x = START_X - (yField * OFS_X);
y = START_Y + yField * OFS_Y;
```

Von dort aus zeichnen wir die Tiles, und verändern nach jedem Tile unsere Position um den jeweiligen Offset.

```
int tile = getTileAt(xField, yField);
drawTile(doubleBuffer, tiles, x, y, tile);
x += OFS_X;
y += OFS_Y;
```

Das Ergebnis ist die typische, diamantförmige Karte.

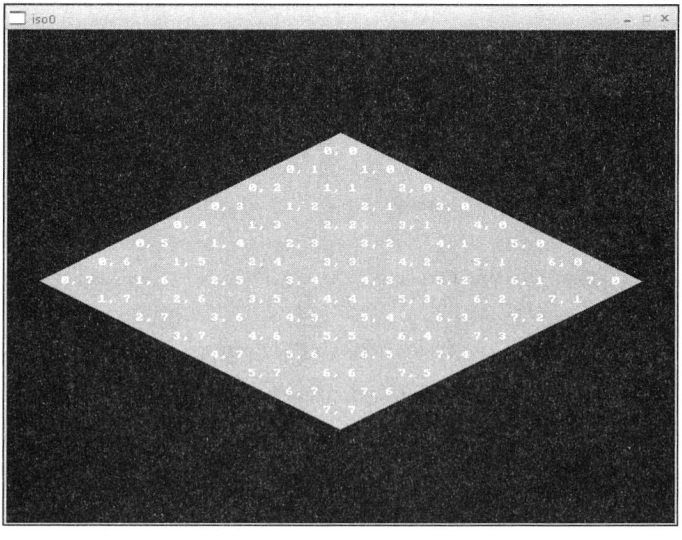


Abbildung 31.6: Iso0.cpp : eine erste isometrische Karte

Das komplette Beispielprogramm ist recht kurz. Anstatt einer richtigen Karte verwenden wir einen Algorithmus, der eine Karte mit Schachbrettmuster simuliert.

```
#include <allegro.h>
#include "util.h"

const int TILE_W = 74;
const int TILE_H = 37;

const int MAP_W = 8;
const int MAP_H = 8;

const int OFS_X = 36;
const int OFS_Y = 18;

#define START_X ((SCREEN_W- TILE_W) / 2)
#define START_Y (100)

int getTileAt(int x, int y) {
    return (x + y) & 1;
}

void drawTile(BITMAP* dest, BITMAP* tile, int x, int y, int index) {
    masked_blit(tile, dest,
                index*TILE_W, 0,
                x, y,
                TILE_W, TILE_H);
}


int main(int , char**) {
    init(640, 480, 60, false);

    BITMAP *tiles = load_bitmap("tiles.tga", NULL);

    clear(doubleBuffer);

    int x, y;
    for (int yField = 0; yField < MAP_H; ++yField) {
        x = START_X - (yField * OFS_X);
        y = START_Y + yField * OFS_Y;
        for (int xField= 0; xField < MAP_W; ++xField) {
            int tile = getTileAt(xField, yField);
            drawTile(doubleBuffer, tiles, x, y, tile);
            textprintf_centre(
                doubleBuffer, font,
                x + TILE_W/2,
                y+ (TILE_H - text_height(font))/2,
```

```
                makecol(255,255,255),
                "%i, %i",
                xField, yField
            );
            x += OFS_X;
            y += OFS_Y;
        }
    }
    show();
    clear_keybuf();
    readkey();

    destroy_bitmap(tiles);

    return 0;
} END_OF_MAIN()
```

Je nach Art der Tiles muss der Offset angepasst werden.

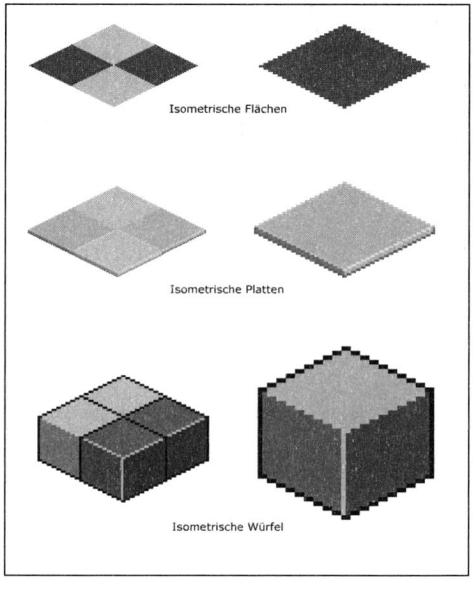


Abbildung 31.7: Arten von Tiles

Das Beispiel `iso1.cpp` zeigt eine aus Platten bestehende isometrische Welt an:

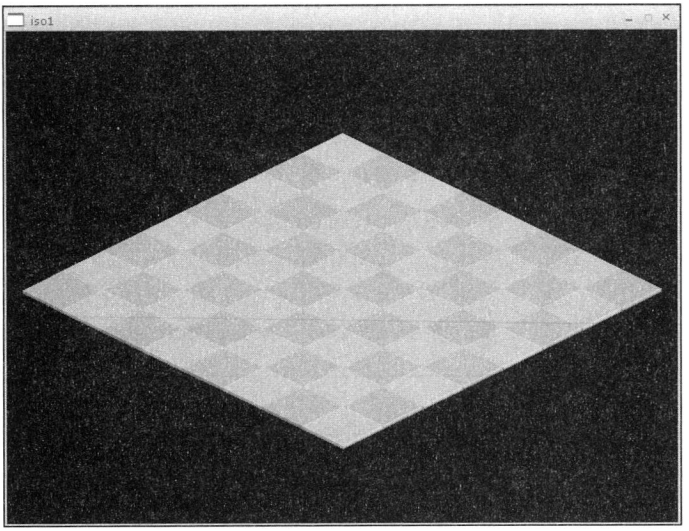


Abbildung 31.8: Iso1.cpp: Platten als Tiles

Und in Beispiel `iso2.cpp` sehen Sie eine aus Würfeln bestehende Welt (siehe Abbildung 31.9).

Sie müssen von Fall zu Fall entscheiden, ob Sie nun lieber Flächen, Platten oder Würfel als Grundlage für Ihre Welt nehmen wollen.

Es gibt eine exzellente Erklärung (in englischer Sprache), die genau beschreibt, worauf man beim Zeichnen von isometrischen Kacheln achten muss. Dieses Tutorial finden Sie im Internet auf der Seite *http://www.indie-rpg.net/pixel-zone/shtml/tut-isometric.shtml*.

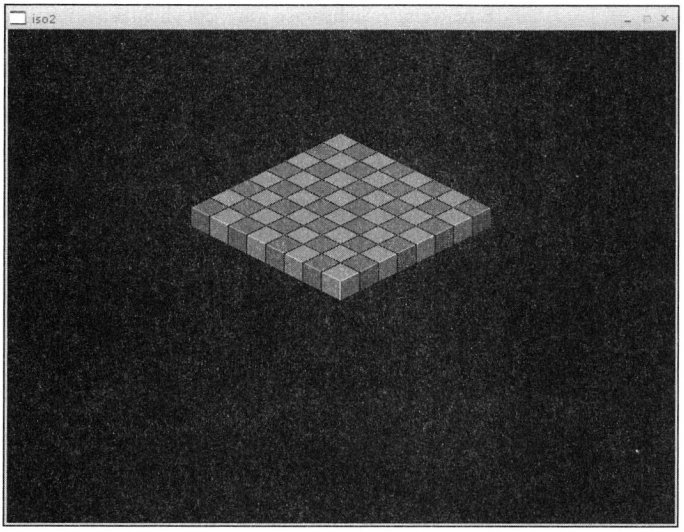


Abbildung 31.9: iso2.cpp: Eine Welt aus Würfeln

Alternative Zeichenroutine

Den Algorithmus, den wir bisher zum Zeichnen der Karte verwendet haben, eignet sich hervorragend für das Anzeigen einzelner Räume. Wenn Ihr Spiel also aus isometrischen Räumen besteht, zwischen denen umgeschaltet wird, dann können Sie diese Methode gefahrlos weiterverwenden.

Allerdings hat diese Methode einige Nachteile:

- Das Bestimmen der Startposition ist recht aufwendig.
- Das Berechnen der Feldkoordinaten aus den Bildschirmkoordinaten ist aufwendig.

Glücklicherweise können wir einen Trick anwenden, durch den sich isometrische Karten wieder beinahe wie normale Karten verwenden lassen.

Wir haben bisher immer die gesamte Karte um 45 Grad gedreht und dann die Karte entsprechend angezeigt.

Eine weitere Sichtweise wäre jedoch anzunehmen, die Karte bestehe aus einem Feld von karoförmigen Kacheln, bei denen jede zweite Reihe um eine halbe Kachel eingerückt ist (siehe Abbildung 31.10).

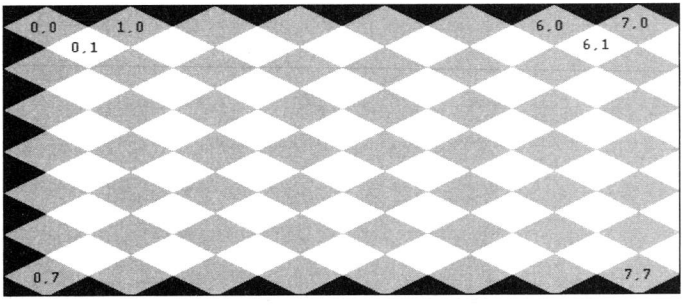


Abbildung 31.10: Eine andere Sichtweise einer isometrischen Karte

Bei dieser Sichtweise werden die Kacheln weiterhin von rechts nach links und von oben nach unten gezeichnet. Allerdings sind die Reihen nun enger beieinander und jede gerade Zeile ist um ein halbes Feld eingerückt.

Sie können auf diese Weise genauso diamantförmige Karten erzeugen, allerdings haben Sie dann auf den Seiten ein paar überschüssige Kacheln.

Abbildung 31.11: Iso-Karte und sichtbarer Bereich

Sie können nun die Karte wie gewohnt zeichnen:

```
int x, y;
y = START_Y;
for (int yField = 0; yField < MAP_H; ++yField) {
    x = START_X;
    if (yField & 1) {
        x += OFS_X;
    }
    for (int xField = 0; xField < MAP_W; ++xField) {
        int tile = getTileAt(xField, yField);
        drawTile(doubleBuffer, tiles, x, y, tile);
        x += TILE_W;
    }
    y += OFS_Y;
}
```

Die Konstanten START_X und START_Y werden wohl meist (0,0) sein – und komplett verschwinden, sobald Sie den normalen, scrollenden Anzeigemechanismus verwenden.

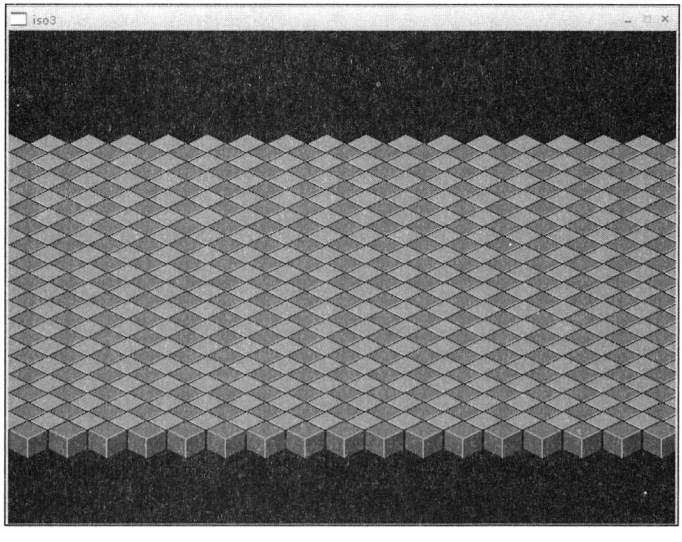


Abbildung 31.12: Iso3.cpp: Rechteckige Iso-Karten

Bildschirm zu Karte

Das Anzeigen der Karte funktioniert schon recht gut – aber wie bekommt man jetzt heraus, auf welches Feld der Karte geklickt wird?

Zwar ist das Vorgehen beim Anzeigen der Karte zu einem Großteil genauso wie bei einer normalen Tile Map, aber bei der Umrechnung der Koordinaten ist leider ein kleiner Zwischenschritt nötig.

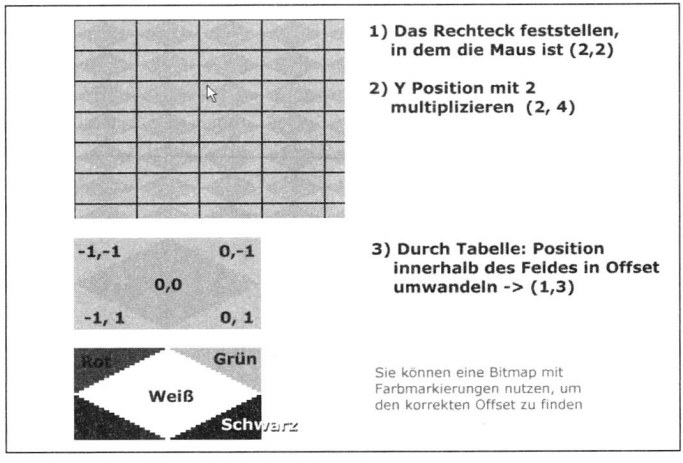


Abbildung 31.13: Von der Maus- zur Tileposition

Der erste Schritt ist analog zur normalen Karte: Sie teilen die Mausposition durch die Breite bzw. Höhe einer Kachel.

Da jedoch die Kacheln bei einer isometrischen Karte im halben Abstand gezeichnet werden, müssen Sie nun den Y-Wert verdoppeln.

Nun berechnen Sie die Position der Maus in dem gerade gefundenen Rechteck und nutzen dies als Index für eine Tabelle.

Stellen Sie sich vor, die farbige Kachel aus Abbildung 31.13 wird über das gefundene Rechteck gelegt. Nun prüfen Sie die Farbe des Bereiches, der unter dem Mauszeiger liegt, und passen die Tile Position dementsprechend an.

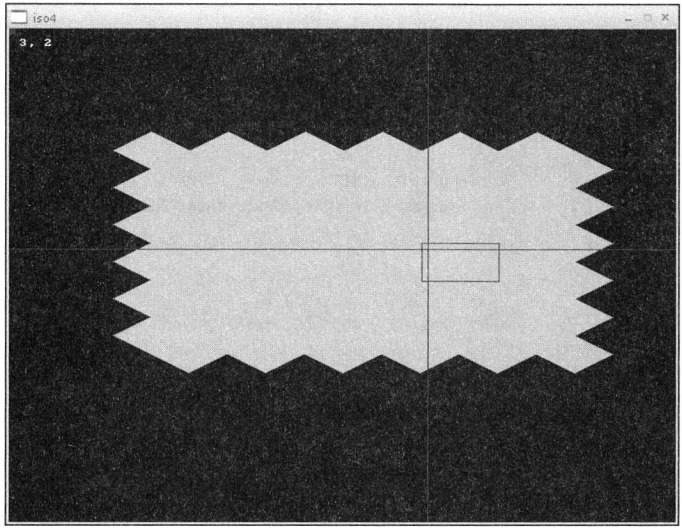


Abbildung 31.14: Iso4.cpp: Berechnen der Kachel unter der Maus

Je nach Art Ihrer Kacheln ist es womöglich besser, die Position des Rechtecks durch die Verwendung von (OFS_X*2) und (OFS_Y*2) zu berechnen, da sich sonst durch die Abweichung einige Fehler einschleichen können.

```
// iso4.cpp
// Berechnung der Kartenposition anhand
// der Mausposition

#include <allegro.h>
#include "util.h"

const int TILE_W = 74;
const int TILE_H = 37;

const int MAP_W = 6;
const int MAP_H = 12;

const int OFS_X = 36;
const int OFS_Y = 18;
```

```
#define START_X (100)
#define START_Y (100)

int getTileAt(int x, int y) {
    return y & 1;
}

void drawTile(BITMAP* dest, BITMAP* tile, int x, int y, int index) {
    masked_blit(tile, dest, index*TILE_W, 0, x, y, TILE_W, TILE_H);
}


int main(int , char**) {
    init(640, 480, 60, false);

    BITMAP *tiles  = load_bitmap("tiles.tga", NULL);
    BITMAP *lookUp = load_bitmap("mask.tga", NULL);

    int red    = makecol(255,0,0);
    int green  = makecol(0,255,0);
    int blue   = makecol(0,0, 255);
    int black  = 0;
    int white  = makecol(255,255,255);


    int tw = OFS_X *2;
    int th = OFS_Y *2;

    while (! key[KEY_ESC]) {
        clear(doubleBuffer);

        int x, y;
        y = START_Y;
        for (int yField = 0; yField < MAP_H; ++yField) {
            x = START_X;
            if (yField & 1) {
                x += OFS_X;
            }
            for (int xField = 0; xField < MAP_W; ++xField) {
                int tile = getTileAt(xField, yField);
                drawTile(doubleBuffer, tiles, x, y, tile);
                x += TILE_W;
            }
            y += OFS_Y;
        }
```

```
        int mx = mouse_x;
        int my = mouse_y;
        hline(doubleBuffer, 0, my, SCREEN_W, makecol(255,0,0));
        vline(doubleBuffer, mx, 0, SCREEN_H, makecol(255,0,0));

        int tileX = (mx - START_X) / tw;
        int tileY = (my - START_Y) / th;
        int ox    = (mx - START_X) % tw;
        int oy    = (my - START_Y) % th;

        rect(doubleBuffer, START_X + tileX * tw,
                           START_Y + tileY * th,
                           START_X + tileX * tw + tw,
                           START_Y + tileY * th + th,
                red
            );

        int col = getpixel(lookUp, ox,oy);
        if (col == red) {
            tileX--;
            tileY--;
        } else if (col == green) {
            tileY--;
        } else if (col == blue) {
            tileX--;
            tileY++;
        } else if (col == black) {
            tileY++;
        }

        textprintf(doubleBuffer, font, 10, 10, white, "%i, %i",
        tileX, tileY);
        rectfill(doubleBuffer, 0, 0, 5, 5, col);

        show();
    }


    destroy_bitmap(tiles);
    destroy_bitmap(lookUp);

    return 0;
} END_OF_MAIN()
```

Den kompletten Quellcode finden Sie natürlich auch auf der beigefügten CD.

32 Effekte

In diesem Kapitel lernen Sie Spezialeffekte zu nutzen. Diese reichen von einem Partikelsystem zur Simulation von Explosionen und Sternensystemen über Lichteffekte bis hin zum Überblenden von Bildern. Effekte sind das Salz in der Suppe eines Spieles. Ein gut eingesetzter Effekt kann für kurzzeitige Überraschung sorgen und dem Spieler ein Gefühl des Staunens vermitteln. Oder er ist so dezent, dass er dem Spieler erst nach wiederholtem Spielen auffällt – oder auch nur im Vergleich mit Spielen, die diesen Effekt nicht haben.

Partikelsysteme

Sternenfelder, Explosionen, wirbelnde Buchstaben und davon fliegende Vögelschwärme – all dies können Sie mit Partikelsystemen simulieren. Die Grundidee ist sehr einfach: Man definiert einen Partikel mit einigen Eigenschaften wie Aussehen, Lebensdauer, Position, Geschwindigkeit und Richtung. Dann definiert man ein Verhalten für diesen Partikel, im einfachsten Falle könnte dies sich auf die Bewegung des Partikels beschränken.

Sternensysteme I

Und dann erzeugt man eine große Menge dieser Partikel und bewundert das Ergebnis. Um zum Beispiel einen bewegten Sternenhimmel zu simulieren, reichen ein paar Zeilen Code aus. Und das Ergebnis lässt sich sehr gut in Weltraumspielen oder auch im Abspann eines Spieles benutzen.

Jeder Stern ist ein einzelner Partikel. Je heller der Stern ist, um so näher erscheint er dem Betrachter. Aus diesem Grund sollte er sich dann auch schneller bewegen als die im Hintergrund befindlichen Sterne. Dies erzeugt einen Parallaxeneffekt, der in Verbindung mit der Farbgebung ein Gefühl der Tiefe vermittelt.

```
class Star {
    float x;
    float y;
    float dx;
```

```
    float dy;

    int  index;

public:
    Star() {
        init();
    }

    void init(int firstTime = TRUE) {
        x = rand() % SCREEN_W;
        if (firstTime) {
            y = rand() % SCREEN_H;
        } else {
            y = - rand() % 20;
        }

        index = rand() % 63 +1;
        dy = index * 0.125;
        if (dy < 0.5) {
            dy = 0.5;
        }
        dx = 0;

    }

    void update() {
        x += dx;
        y += dy;

        if (y > SCREEN_H) {
            init(FALSE);
        }
    }

    void render(BITMAP *bmp) {
        int r = (index / 28);
        if (r < 1) {
            r = 1;
        }
        circlefill(bmp, (int)x, (int)y, r, 128+index);
    }
};
```

Abbildung 32.1: Bewegte Sterne vor einem fixen Hintergrund

Wie Sie sehen können, ist die `Star`-Klasse noch sehr überschaubar. Sie hat nur ein paar wenige Attribute für Position, Geschwindigkeit und den »Index« des Sterns. Der Index entspricht der umgekehrten Entfernung vom Betrachter. Bei einem niedrigen Index ist der Stern weit entfernt, bei einem hohen Index ist er nah am Betrachter.

Die `init()`-Methode weist einem `Star`-Objekt zufällige Werte zu. Die zugewiesene Y-Position hängt davon ab, ob es sich um den ersten Aufruf (`firstTime == TRUE`) oder eine Reinitialisierung handelt.

Beim ersten Aufruf wird der `y` Wert auf eine beliebige, sichtbare Position gesetzt, bei den folgenden Aufrufen wird sicher gestellt, dass der zugewiesene Wert oberhalb des sichtbaren Bereichs liegt.

Die Geschwindigkeit in Y-Richtung wird in Abhängigkeit vom Index ermittelt, um den oben angesprochenen Tiefeneffekt zu erzielen. Die Geschwindigkeit in X-Position ist in diesem Beispiel gleich Null – die Sterne bewegen sich also ausschließlich von oben nach unten über den Bildschirm.

Die update()-Methode, welche die Position des Sterns verändert, überprüft ob sich der Stern bereits außerhalb des sichtbaren Bereichs befindet. Ist dies der Fall, dann wird der Stern neu initialisiert.

Die Anzeige erfolgt mit Hilfe der Funktion circlefill(), die einen ausgefüllten Kreis an der angegebenen Position zeichnet. Wir verwenden an dieser Stelle den Index des Sterns als Farbe – dies hängt damit zusammen, dass das Beispielprogramm einen 8-Bit-Grafikmodus benutzt.

Um nun den Sternenfeldeffekt zu erzeugen wird eine bestimmte Anzahl dieser Sterne erzeugt. Im Beispiel sind es 200, aber Sie können auch gerne 400 oder 800 Sterne benutzen.

```
const int COUNT_STARS = 200;
typedef vector<Star*> StarCollection;
typedef StarCollection::iterator StarIterator;

// und dann im eigentlichen Programm
    StarCollection stars;
    stars.reserve(COUNT_STARS);
    for (int a=0; a < COUNT_STARS; a++) {
        Star *star= new Star();
        stars.push_back(star);
    }
```

Es werden also COUNT_STARS-Sterne erzeugt und im stars-Vector gespeichert. Die Anzeige in der Hauptschleife erfolgt nach genau dem gleichen Schema. Alle Sterne werden erst bewegt (mittels update()) und dann angezeigt.

```
    while (!keypressed()) {

        if (timerCounter) {
            while (timerCounter) {
                blit(bg, doublebuffer, 0, 0, 0, 0,
                                        bg->w, bg->h);
                for (int a=0; a < COUNT_STARS; a++) {
                    stars[a]->update();
                    stars[a]->render(doublebuffer);
                }
                timerCounter--;
            }
            blit(doublebuffer, screen,
                   0, 0, 0, 0,
                   SCREEN_W, SCREEN_H);
        }
    }
```

Übungen

Ändern Sie den Beispielquellcode so ab, dass Sie mit den Cursortasten die Bewegung der Sterne in X-Richtung beeinflussen können. Auch hier sollte die augenscheinliche Nähe zum Betrachter berücksichtigt werden. Die näheren Sterne sollten sich also schneller entlang der X-Achse bewegen als die weiter entfernten Sterne.

Ändern Sie den Quellcode so ab, dass das Beispielprogramm in 16-Bit-Farbtiefe arbeitet.

Sternensysteme II

Beinahe jedes Betriebssystem wird heutzutage mit einer Variante eines Sternenfeld-Bildschirmschoners ausgeliefert. In den meisten Fällen erscheinen diese Sterne in der Mitte, und bewegen sich geradlinig vom Zentrum weg und erzielen dadurch den Eindruck, man würde sich durch ein Meer von Sternen bewegen.

Und diese Beschreibung (Position der Sterne beim Erzeugen und Art der Bewegung) reichen vollkommen aus, um das erste Sternenfeldprogramm zu einem Pseudo-3D-Sternenfeld umzuwandeln.

Wir haben bisher nur die Position des Sterns in Abhängigkeit von seiner Geschwindigkeit geändert. Nun werden wir auch die Geschwindigkeit des Sterns verändern, damit wieder ein Parallaxeneffekt (Gegenstände, die näher am Betrachter sind, scheinen sich schneller zu bewegen) entsteht.

```
class Star {
    float x;
    float y;
    float dx;
    float dy;

    int  index;

public:
    Star() {
        init();
    }

    void init(int firstTime = TRUE) {
        x = SCREEN_W / 2 + 10;
        y = SCREEN_H / 2 + 10;
```

```
        // Allegros Winkelsystem geht von
       // 0 bis 255
        float angle = (rand() % 255);
        dx = fixtof(fixcos(ftofix(angle)));
        dy = fixtof(fixsin(ftofix(angle)));
        index = 32 + rand()%32;
    }

    void update() {
        x += dx;
        y += dy;

        dx *= 1.0 + 0.5 / (float) (rand()%9+1);
        dy *= 1.0 + 0.5 / (float) (rand()%9+1);;

        if (y > SCREEN_H || y < 0 || x > SCREEN_W || x < 0) {
            init(FALSE);
        }
    }

    void render(BITMAP *bmp) {
        putpixel(bmp, (int)x, (int)y, 128+index);
    }
};
```

Bei den Attributen der Klasse hat sich nicht viel getan, aber in der `init()`-Methode ist nun der Startpunkt und die Farbe fix, und die Richtung wird anhand des »Austrittwinkels« des Sterns berechnet.

In der `update()`-Methode wird die Geschwindigkeit des Sterns immer leicht erhöht, damit ein Stern der schon länger auf dem Schirm (und damit also auch näher am Betrachter) ist, sich auch schneller bewegt als ein Stern, der gerade erst erzeugt wurde.

Der Rest des Programms bleibt unverändert.

Übungen

- ✔ Erlauben Sie es dem Benutzer den Startpunkt der Sterne mit den Cursortasten zu verschieben.
- ✔ Ändern Sie das Programm so ab, dass sich nicht nur die Geschwindigkeit, sondern auch die Größe des Sterns mit der Zeit verändert.
- ✔ Ändern Sie das Programm so ab, dass anstatt eines einzelnen Punktes eine Linie von der derzeitigen Position des Sterns zu seiner vorherigen Position gezeichnet wird.

Raketentriebwerke

Mit Hilfe von Partikelsystemen kann man auch sehr gut den Rückstossstrahl von Raketentriebwerken simulieren. Der dafür nötige Code ist dem bisherigen Code sehr ähnlich, nur muss der Austrittswinkel der Partikel der Flugrichtung der Rakete angepasst werden.

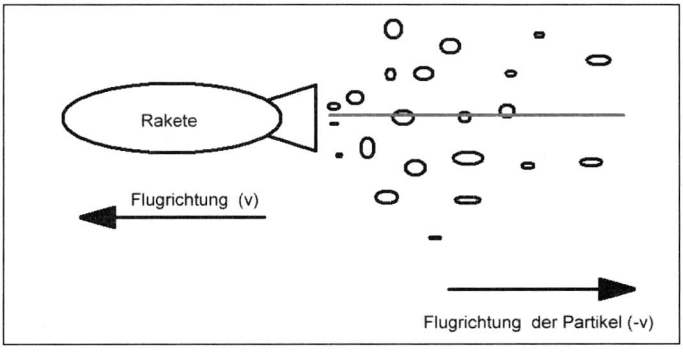


Abbildung 32.2: Partikelsystem für eine Rakete

Die Partikel der Rakete bewegen sich entgegengesetzt der Richtung der Rakete. Um also die Richtung eines Partikels berechnen zu können, muss der Flugwinkel der Rakete bekannt sein.

Ist von der Rakete die Bewegung in X- und Y-Richtung bekannt, dann kann man mit Hilfe des Arcus Tangens den dazugehörigen Winkel berechnen.

```
winkel = atan2( dy * -1.0, dx *-1.0)
```

Beachten Sie hierbei, dass die Reihenfolge der Parameter `dy, dx` ist – die beiden Werte in umgekehrter Reihenfolge zu übergeben ist ein häufiger Fehler.

Sobald der Winkel bekannt ist, kann man ihn leicht variieren, indem man einen Zufallswert aufaddiert. Sobald der endgültige Winkel feststeht, kann mit den bekannten Funktionen die Positionsänderung pro Frame für den Partikel berechnet werden.

Für alle Winkelfunktionen werden die Allegro Funktionen benutzt, die anstatt den resultierenden Wert über Näherungsalgorithmen zu berech-

nen (wie die entsprechenden Standardfunktionen), den Wert direkt aus einer Tabelle lesen.

Neben dem Austrittswinkel ist natürlich auch die Startposition des Partikels von Bedeutung. Und die ist natürlich abhängig von der Position der Rakete. Aus diesem Grund braucht jeder Partikel eine Referenz auf die Rakete, zu der er gehört. Dies erlaubt es ihm, bei der Initialisierung die Position und Richtung der Rakete zu berücksichtigen.

Lebensdauer

Bisher wurden die Partikel neu initialisiert, sobald sie den Bildschirm verlassen haben. Bei einem Raketenrückstoss liegt die Sache allerdings anders – dieser sollte auf einen relativ kleinen Bereich hinter der Rakete begrenzt sein. Die Lösung liegt in der Einführung einer Lebensdauer für den Partikel. Jeden Frame wird dieser Wert verringert und sobald er bei Null ankommt, wird der Partikel neu initialisiert.

```
class SmokeParticle {
    float x;
    float y;
    float dx;
    float dy;
    int   togo;

    int  index;

    Rocket *parent;

public:
    SmokeParticle(Rocket *parent) {
        this->parent = parent;
        init();
    }

    void init(int firstTime = TRUE) {
        x = parent->x;
        y = parent->y;

        float angle = fixtof(fixatan2(
                            ftofix(parent->dy *-1.0),
                            ftofix(parent->dx * -1.0)
                        ) +itofix((rand() % 40) -20)
                    );
```

```
        dx = fixtof(fixcos(ftofix(angle))) * 0.5;
        dy = fixtof(fixsin(ftofix(angle))) * 0.5;
        index = 32 + rand()%32;

        togo = rand()% 20 + 10;
    }

    void update() {
        x += dx;
        y += dy;

        dx *= 1.0 + 0.5 / (float) (rand()%9+1);
        dy *= 1.0 + 0.5 / (float) (rand()%9+1);;

        togo--;
        if (togo <= 0) {
            init(FALSE);
        }
    }

    void render(BITMAP *bmp) {
        putpixel(bmp, (int)x, (int)y, 128 +togo*2);
    }
};
```

Die Rakete ist im Beispiel nur durch Position und Richtung definiert:

```
struct Rocket {
    float x, y;
    float dx, dy;
};
```

Die Bewegung der Rakete wird im Main-Loop von Hand durchgeführt:

```
        rocket.x += rocket.dx;
        rocket.y += rocket.dy;
        if (rocket.x < 0) {
            rocket.x = SCREEN_W;
        }
```

Wenn Ihnen dieser Code bekannt vorkommt, dann haben Sie nicht ganz unrecht. Er sieht dem `update()`-Code der ersten Sternenfeldversionen doch sehr ähnlich.

Übungen

- ✔ Machen Sie sich die Ähnlichkeit der Rakete und des Partikels zu Nutze, und erstellen Sie eine gemeinsame Basisklasse für Rakete und Partikel.
- ✔ Erstellen Sie eine Zeichenfunktion für die Rakete. Benutzen Sie hierbei die `rotate_sprite()`-Funktion.
- ✔ Verändern Sie den Code so, dass mehrere Raketen auf beliebigen Bahnen über den Schirm fliegen.

Explosionen

Sie können Partikelsysteme nutzen, um für ein Spiel Explosionen entweder in Echtzeit zu erzeugen oder sie vorab zu berechnen und dann die Animation als eine Reihe von Bildern zu speichern.

Sie können sogar ein paar Explosionsanimationen vorab berechnen und dann diese benutzen, um eine größere Explosion in Echtzeit zu erzeugen.

All diese Varianten haben die gleiche grundlegende Funktionsweise. Vom Zentrum der Explosion werden Partikel nach außen geschleudert. Jeder Partikel hat eine gewisse Lebensdauer, und wenn diese überschritten ist, dann verschwindet entweder der Partikel oder er teilt sich in mehrere Unterpartikel, die dann wiederum selbst eine Lebensdauer haben.

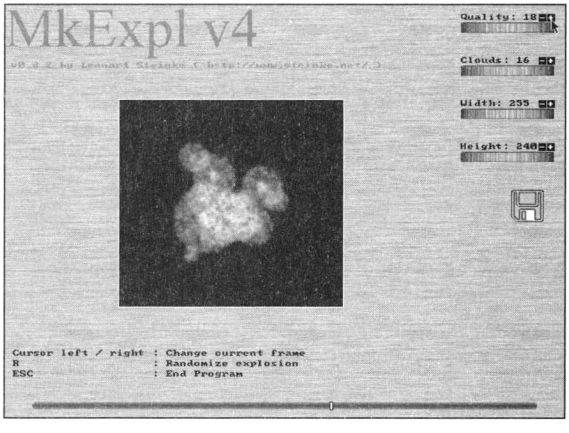


Abbildung 32.3: MkExpl v4 – ein Programm, um Explosionen zu erstellen

Das Programm MkExpl benutzt Partikel, die neben der Position und Geschwindigkeit auch eine »Dichte« haben. Je höher die Dichte, umso mehr Pixel werden in einem festgelegten Radius um den Mittelpunkt des Partikels herum erzeugt. Dies funktioniert ähnlich wie die Sprühdosenfunktion (»Airbrush«) in einem Malprogramm. Um weiche Übergänge zu erhalten, wird dann die resultierende Bitmap weichgezeichnet.

```
typedef struct {
    int x, y, r, d;
    float dx, dy, dr, dd;

    void *next;
    void *prev;
    void *parent;
} Particle;
```

Variable	Bedeutung
x, y	Position
dx, dy	Positionsänderung pro Frame
r	Radius
dr	Radiusänderung pro Frame
d	Dichte der Explosion
dd	Änderung der Explosionsdichte pro Frame

Tabelle 32.1: Komponenten der MkExpl-Partikel-Struktur

MkExpl ist ein C-Programm und kann aus diesem Grund nicht auf die STL zurückgreifen. Die Partikel enthalten aus diesem Grund Variablen, um eine verkettete Liste aufzubauen.

Da das Programm jeden beliebigen Frame der Animation jederzeit neu zeichnen können muss, werden alle Werte ausgehend vom Initialwert berechnet. Auch hat jeder Frame seinen eigenen Ausgangswert für den Pseudozufallszahlengenerator, um sicherzustellen, dass auch jedes Mal die gleiche Folge von »Zufallszahlen« für den jeweiligen Frame erzeugt wird.

```
void drawFrame(int frameNum, ParticleSystem* ps) {
    FrameInfo* frame = &frames[frameNum];
    Particle*  p     = NULL;
    int        x,y,r,d,a;
    int        fm = (frameCount/4);

// Den Frame nur updaten wenn es sein muss
    if (frame->needsUpdate) {
        clear(frame->buffer);
        // Zufallszahlen Startwert setzen
        srand(frame->randInit);
        // So oft weichgezeichnet werden soll
        for (a=0; a < blurCount; a++) {
            // Für alle Partikel
            p = ps->first;
            while (p != NULL) {
                // Berechnen der Position,
                // Größe und Dichte in diesem Frame
                x = p->x + p->dx * frameNum;
                y = p->y + p->dy * frameNum;
                r = p->r + p->dr * frameNum;

                if (frameNum >= fm) {
                    d = MAX(0, 90 - (100 * frameNum)/frameCount);
                } else {
                    if (frameNum > 0) {
                        d = p->d + (((90- p->d) /frameNum) / fm);
                    } else {
                        d = p->d;
                    }
                }
                // Aufruf der Airbrush Funktion
                spray(frame->buffer, x,y,r , d/2, 255);
                spray(frame->buffer, x,y, r/4, d, 255);
                spray(frame->buffer, x,y,r , d/4, 128);
                p = p->next;
            }
            // Weichzeichnen
            blurFrame(frame);
        }
        frame->needsUpdate = 0;
    }
}
```

MkExpl v4 liegt in einer nicht ganz vollständigen Version vor, ist aber die einzige Variante, die mit Allegro geschrieben wurde. Version 3.0 erzeugt optisch ansprechendere Explosionen, ist aber nur für DOS verfügbar.

Wenn Sie mit dem Quellcode von MkExpl experimentieren möchten, dann würde ich empfehlen ein paar »dunkle Partikel« hinzuzufügen, welche die Gleichmäßigkeit des Explosionszentrums unterbrechen und damit realistischere Grafiken erzeugen.

Licht- und Transparenzeffekte

Stellen Sie sich einen normalen Dungeon in Ihrem Spiel vor. Sie haben großartige Tile Grafiken für den Boden, die Wände sehen sehr detailliert aus, und Ihre Monster sind schauderlich.

Was könnte man mehr wollen? Spulen wir die Szene zurück, und stellen sie uns noch mal vor. Nur diesmal mit kleinen Änderungen.

Der Dungeon ist ein düsteres Licht gehüllt, nur ein paar wenige Fackeln erschaffen flackernde Inseln des Lichts. Schemenhaft sehen Sie die Monster die im Schatten herum schleichen...

Mit ein paar Lichteffekten können Sie Ihrem Spiel eine besondere Note geben – nicht nur das, die Grafiken wirken interessanter und lebendiger. Und es geht so einfach.

Allegros Halbtransparenzfunktionen

Allegro stellt Ihnen Funktionen zur Verfügung, mit denen Sie Grafiken halbtransparent anzeigen können. Dies erlaubt es Ihnen Menüs zu entwerfen, bei denen man die Spielgrafik noch durchschimmern sehen kann. Sie können mit diesen Funktionen auch Wassereffekte erzeugen.

Der Nachteil ist, dass zum Berechnen der endgültigen Grafik ein Lesezugriff auf beide beteiligten Bitmaps nötig ist.

Das Anzeigen eines halbtransparenten Bildes erfolgt also in diesen Schritten:

```
Für alle Pixel
    Hole die RGB Werte des Ausgangspixels
    Hole die RGB Werte des Zielpixels
    Kombiniere die Werte entsprechend der Transparenzstufe
    Schreibe den resultierenden Farbwert auf die Ziel Bitmap
```

Da ein Lesezugriff auf Videobitmaps sehr zeitaufwendig ist, sollten sich beide Bitmaps entweder im System- oder Hauptspeicher befinden.

Zur Anzeige von halbtransparenten Sprites werden Sie vor allem diese beiden Funktionen benötigen:

Funktion	`void set_trans_blender(int r, int g, int b, int a);`
Beschreibung	Setzt die Parameter für die Transparenz-Zeichen-Funktionen. Die Parameter `r`,`g` und `b` werden ignoriert. Der Parameter `a` gibt die Stufe der Transparenz an, wobei 0 für komplett durchsichtig, 255 für komplett sichtbar und 128 für 50% Transparenz steht.
Funktion	`void draw_trans_sprite(BITMAP *bmp, BITMAP *sprite, int x, int y);`
Beschreibung	Zeigt die Bitmap `sprite` an Position `x`,`y` mit der durch `set_trans_blender()` festgelegten Transparenz an.

Tabelle 32.2: set_trans_blender() und draw_trans_sprite()

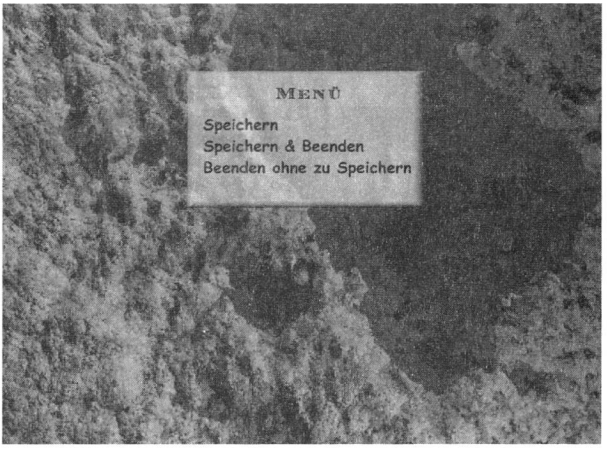


Abbildung 32.4: Anzeige des Menüs mit 20% Transparenz

Wenn man sich diese Funktionen so anschaut, dann werden Ihnen sicherlich sofort einige Fragen durch den Kopf schießen:

- Warum muss ich `r,g` und `b` übergeben, wenn diese Werte doch nicht benutzt werden?
- Warum muss ich für jedes Sprite den Transparenzwert neu setzen?
- Warum wird die Transparenzstufe nicht bei `draw_trans:sprite()` übergeben?

Die Lösung liegt in der internen Verwaltung dieser Überblendfunktionen. Die derzeit aktive Funktion jeder einzelnen Gruppe (transparent, beleuchtet oder alpha) wird in einem Funktionszeiger gespeichert. Das bedeutet, dass alle Überblendfunktionen die gleiche Signatur haben müssen.

Wir werden nun das Beispielprogramm aus dem Partikelsystemteil wiederverwenden um einige Geister durch eine Schlossruine zu scheuchen.

Abbildung 32.5: Geister in der Schlossruine

Am Code hat sich nicht sehr viel geändert, nur die Bestimmung der Richtung und die Zeichenfunktion wurden leicht angepasst.

```
void init(int firstTime = TRUE) {
    x = -40 - rand() % 200;
    y = rand() % SCREEN_H;

    // Die Geister sollen von links nach rechts
    // fliegen
    dy = (float) (rand() % 10 +1) / 6.0 - 1.2;
    dx = (float) (rand() % 10 +1) / 4.2;

    trans = rand() % 128 + 64;
}

void render(BITMAP *bmp) {
    set_trans_blender(0,0,0,trans);
    draw_trans_sprite(bmp, ghost, (int) x, (int)y);
}
```

Das vollständige Programm finden Sie auf der CD.

Lichteffekte

Lichteffekte basieren auf der gleichen Idee wie transparente Sprites. Nur anstatt `set_trans_blender()` zu benutzen, wird `set_add_blender()` benutzt.

Funktion	`set_add_blender(int r, int g, int b, int a)`
Beschreibung	Setzt den Additions-Überblendmodus.

Tabelle 32.3: set_add_blender()

Beim Additions-Überblendmodus werden die Zielpixel wie folgt berechnet:

```
Für alle Pixel
    Lies RGB Werte des Ausgangpixels
    Lies RGB Werte des Zielpixels
    Berechen neuen Zielpixel mit
     Zielpixel = ZielPixel + AusgangPixel * a / 255
```

Einen Lichtkegel können Sie mit ein paar Zeilen Code über den Schirm gleiten lassen:

```
int main(int, char**) {

    srand(time(NULL));
    allegro_init();
    set_color_depth(16);
    if (set_gfx_mode(GFX_AUTODETECT, 800, 600, 0, 0) != 0) {
        set_color_depth(15);
        if (set_gfx_mode(GFX_AUTODETECT, 800, 600, 0, 0) != 0) {
            allegro_message("Unable to set GFX mode");
            exit(-1);
        }
    }
    install_timer();
    install_keyboard();

    /* Timer Funktionen und Variablen locken */
    LOCK_FUNCTION(timerCounterUpdater);
    LOCK_VARIABLE(timerCounter);
    /* Die logische Framerate setzen */
    install_int_ex(timerCounterUpdater, BPS_TO_TIMER(30));

    BITMAP* bg    = load_bitmap("skyline.tga", NULL);
    BITMAP* light = load_bitmap("light.tga", NULL);

    BITMAP *doublebuffer = create_bitmap(SCREEN_W, SCREEN_H);


    clear_keybuf();
    timerCounter = 0;
    set_add_blender(0,0,0, 128);
    float x,y, dx, dy;
    x = (SCREEN_W - light->w) / 2;
    y = (SCREEN_H - light->h) / 2;
    dy = 3.4;
    dx = 4.2;
    while (!keypressed()) {

        if (timerCounter) {
            while (timerCounter) {
                blit(bg, doublebuffer, 0, 0, 0, 0, bg->w, bg->h);
                draw_trans_sprite(doublebuffer, light, (int)x,
              (int)y);
```

```
                x+=dx;
                y+=dy;
                if (x + light->w >= SCREEN_W) {
                    x = SCREEN_W-light->w;
                    dx *=-1;
                } else if (x <= 0) {
                    x = 0;
                    dx *=-1;
                }

                if (y + light->h >= SCREEN_H) {
                    y = SCREEN_H - light->h;
                    dy *=-1;
                } else if (y <= 0) {
                    y = 0;
                    dy *=-1;
                }
                timerCounter--;
            }
            blit(doublebuffer, screen, 0, 0, 0, 0, SCREEN_W,
           SCREEN_H);
        }
    }

    destroy_bitmap(light);
    destroy_bitmap(doublebuffer);
    return 0;
}
END_OF_MAIN()
```

Das Ergebnis sieht dann in etwa so aus wie in Abbildung 32.6.

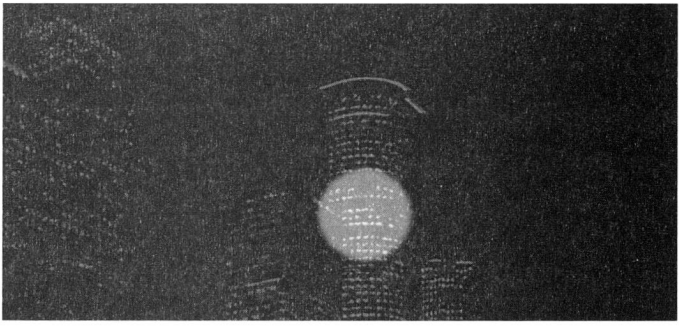


Abbildung 32.6: Lichtkegel in einer Skyline

FBlend – Überblenden auf die schnelle Art

FBlend ist eine Zusatzbibliothek für Allegro, die optimierte und damit sehr schnelle Funktionen für Überblendeffekte zur Verfügung stellt. Es ist geplant, die FBlend-Routinen in einer der nächsten Allegro Versionen zu integrieren, aber bis dahin ist es nötig, auf die externe Version zurückzugreifen.

Falls Sie sich fragen, ob es sich lohnt auf FBlend umzusteigen, dann ist die Antwort ein klares: »Aber sicher doch!«.

Die FBlend-Funktionen sind in der Regel mehr als doppelt so schnell wie die entsprechende Allegro Funktion. In einigen Fällen kann FBlend sogar bis zu 14 mal schneller sein.

FBlend installieren

FBlend lässt sich auf die gleiche Weise installieren wie Allegro. Auf einem Windows Rechner mit MinGW sehen die typischen Schritte etwa so aus:

```
fix mingw32
make
make install
```

Und unter Linux:

```
unzip -a fblend.zip
./fix.sh
make
make install
```

Der –a Parameter beim Aufruf von `unzip` ist unbedingt erforderlich, damit die Files vom DOS Textformat in das Unix-übliche Format konvertiert werden.

Was kann FBlend?

FBlend gliedert sich in drei Teilabschnitte:

- ✔ Bitmap-Überblendroutinen
- ✔ Rechteck-Überblendroutinen
- ✔ Bitmapskalierung

Die Überblend-Routinen kommen immer in zwei Varianten, eine für eine Addition der Farbwerte und eine für eine reine Transparenz. Die Beta 0.5Version hat ebenso Überblendungsroutinen im Subtraktionsverfahren.

Klingt schön und gut, aber was hat man nun davon?

Die »normale« Transparenzroutine ist einfach zu verstehen: Sie zeigt eine Bitmap einfach mit einer beim Aufruf festgelegtem Transparenzstufe an. Allerdings deutlich schneller als es Allegro derzeit tut.

Bei den anderen Routinen verhält es sich ähnlich. Sie funktionieren beinahe identisch, nur hat FBlend bei der Geschwindigkeit die Nase vorn.

Es ist geplant die FBlend-Routinen in die Allegro-Bibliothek direkt einzubauen. Ab diesem Zeitpunkt kommen dann alle Allegro-Programme in den Genuss der optimierten Routinen.

33 Seitenübergänge

In Ihrem Spiel werden Sie häufig von einer Anzeige zur anderen wechseln. Sei es vom Eingangsbild zum Hauptmenü oder vom »Game Over«-Schirm zur Bestenliste. In diesem Kapitel lernen Sie, wie Sie die Übergänge zwischen den Bildern möglichst fließend gestalten können.

Es gibt unzählige Möglichkeiten zwei Bilder ineinander übergehen zu lassen. Und in diesem Kapitel gehen wir auf einige dieser Möglichkeiten ein.

Grundsätzlich gibt es zwei Methoden:

- ✔ Direkte Übergänge (Crossfade)
- ✔ Übergänge über eine leere Seite (Fade Out/Fade In)

Beim direkten Übergang erscheint das neue Bild direkt über oder in dem aktuellen Bild. Es wird zum Beispiel einfach von oben über das derzeitige Bild geschoben. Dies wird in Abbildung 1 gezeigt. Das zweite Bild wird von rechts unten nach links oben diagonal eingeblendet.

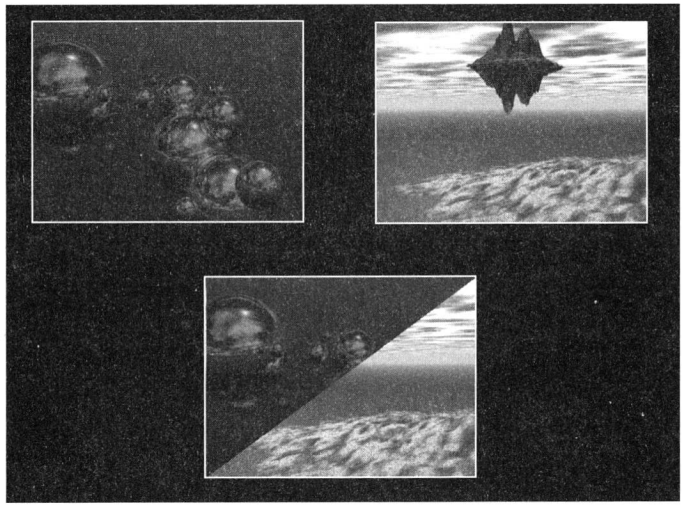


Abbildung 33.1: Ein diagonaler Crossfade

Beim Übergang über ein leeres Bild wird erst ein Bild ausgeblendet (auf welche Weise auch immer) und dann das neue Bild eingeblendet (wieder auf beliebige Weise). Der Vorteil bei dieser Methode ist, dass Sie immer entweder einen definierten Ausgangspunkt (für das Einblenden) oder einen definierten Endpunkt (für das Einblenden) haben. Das erlaubt es Ihnen, diese Methoden beliebig zu kombinieren. Ein Beispiel für das Ein- und Ausblenden finden Sie in Abbildung 33.2.

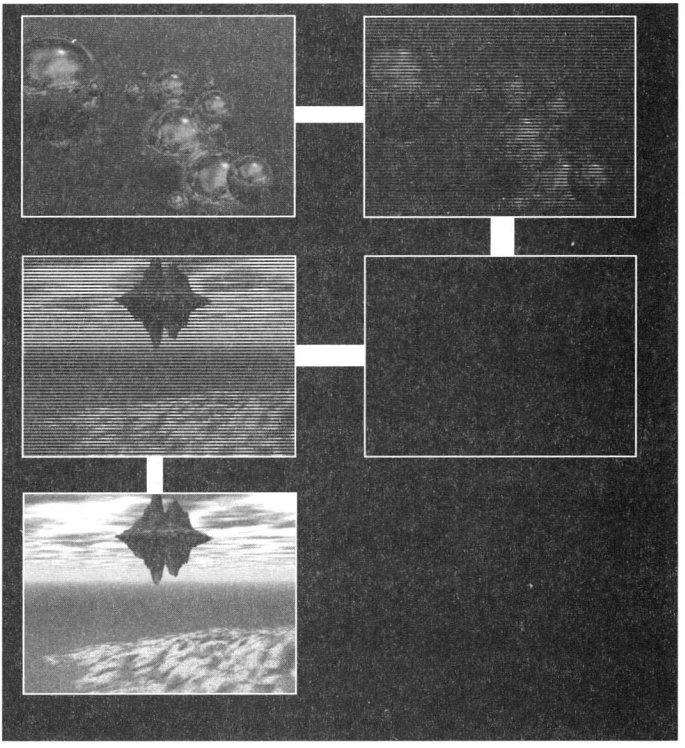


Abbildung 33.2: Aus- und Einblenden eines Bildes

Ein- und Ausblenden

Da diese Methode mit weniger Aufwand zu implementieren ist, werfen wir zuerst einen Blick auf diese Methode. Wie oben geschrieben, ist der große Vorteil, dass man die Fade-Methoden beliebig kombinieren kann. Also sollte es unser Ziel sein, diese Funktionalität auch in unserem Code zu berücksichtigen. Aufgrund der Besonderheit des Fades (entweder Ausgangsbild oder Endbild ist immer ein schwarzer Bildschirm) können wir sogar für das Ein- und Ausblenden die gleiche Methode nutzen.

```
class Fade {
public:
    Fade();
    virtual ~Fade();
    virtual void fade(BITMAP* src, BITMAP* dst,
                                double percent) = 0;
};
```

Diese Klasse legt nur die Schnittstelle für Überblendeffekte fest. Die eigentliche Überblendroutine ist komplett virtuell, das heißt: in dieser Klasse nicht vorhanden. Da es diese Methode nicht gibt, kann die Klasse auch nicht direkt verwendet werden. Erst muss in abgeleiteten Klassen die fehlende Funktion eingebaut werden.

Die `fade()`-Methode funktioniert nach einem recht einfachen Prinzip: sie nimmt die `src`-Bitmap und stellt sie entweder unverändert dar (`percent == 0.0`), komplett schwarz (`percent == 1.0`) oder auf einer beliebigen Stufe dazwischen. Bei `percent == 0.5` ist das Ausblenden zum Beispiel zur Hälfte beendet.

Zum Testen der Überblendklassen brauchen wir nun noch eine Funktion, welche den Aufruf der `fade()`-Methode des jeweiligen Objektes übernimmt.

```
void fade(BITMAP * from, BITMAP *to, Fade *fade, int ticks) {
    double percent;
    int targetTime   = timerCounter + ticks;
    int curTime      = timerCounter;
    int needsRefresh = FALSE;

    while (timerCounter < targetTime && !key[KEY_ESC]) {
        if (curTime <=timerCounter) {
            percent = (double) (ticks - (targetTime - curTime)) /
           (double) ticks;

            curTime      = timerCounter+1;
            needsRefresh = TRUE;
```

```
        }
        if (needsRefresh) {
            needsRefresh = FALSE;
            if (percent <= 0.5) {
                fade->fade(from, doubleBuffer, percent*2.0);
            } else {
                fade->fade(to, doubleBuffer, 1.0 - (percent-
               0.5)*2.0);
            }
            blit(doubleBuffer, screen, 0, 0, 0, 0, doubleBuffer->w,
           doubleBuffer->h);
        }
    }
}
```

Diese Aufgabe übernimmt die Funktion `fade()` im Hauptprogramm. Ihr werden die Bilder übergeben, zwischen denen mittels einer einfachen Blende umgeschaltet werden soll und wie viele logische Zyklen es dauern soll. Sie können hier auch sehen, auf welche Weise Sie die `Fade`-Klasse sowohl zum Einblenden als auch zum Ausblenden nutzen können.

Nun haben wir eine abstrakte Basisklasse und eine Testfunktion – was noch fehlt, sind die Überblend-Klassen.

StripeFade

Die `StripeFade`-Klasse blendet auf Schwarz über, indem erst vom linken und dann vom rechten Rand schwarze Balken frei wählbarer Höhe in das Bild hineingeschoben werden.

```
// ---------------------------------------------------
// -- StripeFade
// ---------------------------------------------------

class StripeFade : public Fade {
    int height;
public:
    StripeFade();
    StripeFade(int h);
    virtual ~StripeFade();
    virtual void fade(BITMAP* src, BITMAP* dst, double
                                                percent);
};

StripeFade::StripeFade() : height(1) {
}
```

```
StripeFade::StripeFade(int h) : height(h) {
}
StripeFade::~StripeFade() {
}
void StripeFade::fade(BITMAP* src, BITMAP* dst, double percent) {
    blit(src, dst, 0,0,0,0, dst->w, dst->h);
    if (percent <= 0.5) {
        percent *= 2;
        int h = height*2;
        int width = (int) (SCREEN_W * percent);
        for (int y=0; y < SCREEN_H; y+=h) {
            rectfill(dst, 0, y, width, y+height-1, 0);
        }
    } else {
        percent -= 0.5;
        percent *= 2;
        int pos = SCREEN_W - (int) (SCREEN_W * percent);
        for (int y=0; y < SCREEN_H; y+=height) {
            rectfill(dst, 0, y, SCREEN_W, y+height-1, 0);
            y+=height;
            rectfill(dst, pos, y, SCREEN_W, y+height-1, 0);
        }
    }
}
```

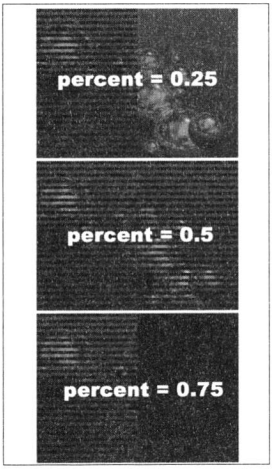


Abbildung 33.3: Verlauf des Überblendens

SymmetricStripeFade

Der SymmetricStripeFade ist dem gerade besprochenen StripeFade sehr ähnlich, nur fahren diesmal die Streifen von beiden Seiten gleichzeitig ins Bild.

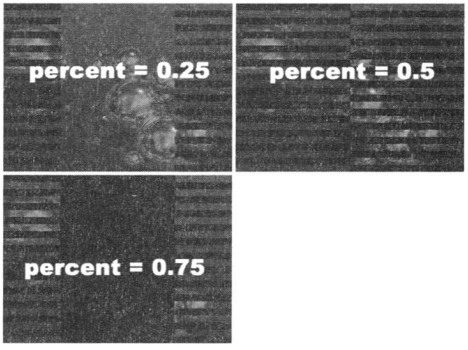


Abbildung 33.4: Der symmetrische StripeFade

Der Quellcode ähnelt auch sehr stark dem des letzen Beispiels, ist aber sogar noch etwas einfacher.

```
// ----------------------------------------------------
// -- SymmetricStripeFade
// ----------------------------------------------------
SymmetricStripeFade::SymmetricStripeFade() : height(1) {
}

SymmetricStripeFade::SymmetricStripeFade(int h) : height(h) {
}
SymmetricStripeFade::~SymmetricStripeFade() {
}
void SymmetricStripeFade::fade(BITMAP* src, BITMAP* dst, double
percent) {
    blit(src, dst, 0,0,0,0, dst->w, dst->h);
    int w = SCREEN_W * percent;
    for (int y=0; y < SCREEN_H; y+=height) {
        rectfill(dst, 0, y, w, y+height-1, 0);
        y+=height;
        rectfill(dst, SCREEN_W-w, y, SCREEN_W, y+height-1, 0);
    }
}
```

Die Prozentzahl entspricht dem Verhältnis der Länge der eingeblendeten Balken zur Breite der Bitmap. Sobald die Länge der sichtbaren Balken feststeht, werden in einer Schleife die Balken von oben nach unten gezeichnet.

StretchFade

Die Stretchblende wird gerne und häufig in Spielen eingesetzt. Es ist ein leicht zu implementierender Effekt, der gut aussieht: Das angezeigt Bild wird immer kleiner, bis es schließlich nicht mehr sichtbar ist.

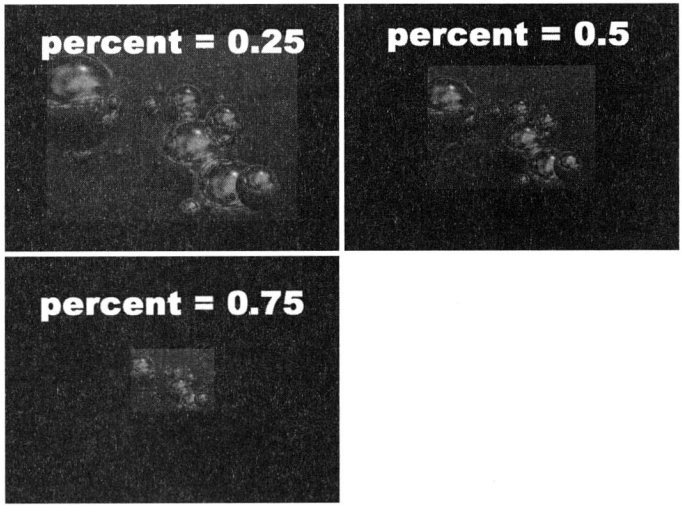


Abbildung 33.5: Der StretchFade: Das Bild verschwindet in der Schirmmitte

Die `fade()`-Routine hat nur sehr wenig zu tun. Zuerst muss die aktuelle Höhe und Breite berechnet werden, dann werden die X- und Y-Koordinaten berechnet, um ein Bild mit der aktuellen Größe zentriert anzuzeigen. Ein Aufruf der `stretch_blit()`-Funktion erledigt den Rest.

```
// ---------------------------------------------------
// -- StretchFade
// ---------------------------------------------------
```

```
StretchFade::StretchFade() {
}

StretchFade::~StretchFade() {
}
void StretchFade::fade(BITMAP* src, BITMAP* dst, double percent) {
    int w = dst->w * (1.0 - percent);
    int h = dst->h * (1.0 - percent);
    int x = (dst->w - w) / 2;
    int y = (dst->h - h) / 2;
    clear(dst);
    stretch_blit(src, dst, 0,0,src->w, src->h, x,y, w, h);
}
```

Wenn Sie die Routine etwas abändern, können Sie weiter interessante Effekte erzielen. Versuchen Sie zum Beispiel einmal ein Bild unterschiedlich schnell in X- und Y-Richtung schrumpfen zu lassen.

ScrollFade

Beim ScrollFade wird das Bild nach rechts aus dem sichtbaren Bereich hinausgeschoben.

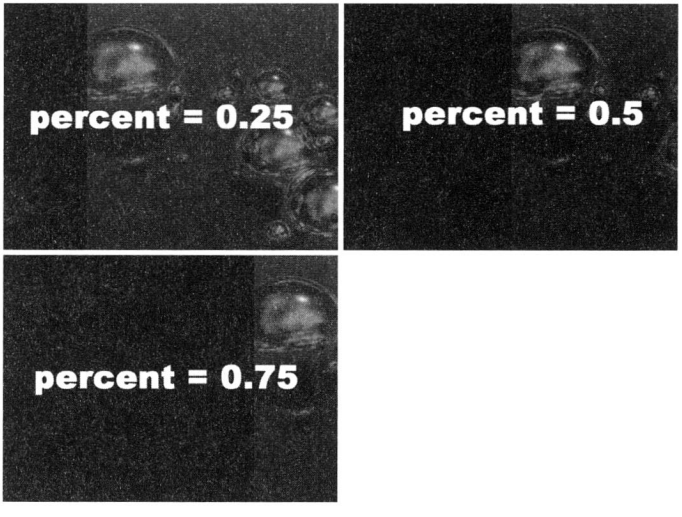


Abbildung 33.6: Beim ScrollFade wird das Bild weggeschoben

Der Code für diesen Effekt ist wieder denkbar einfach: Es wir nur die Startposition des anzuzeigenden Bildes geändert.

```
// ---------------------------------------------------
// -- ScrollFade
// ---------------------------------------------------
ScrollFade::ScrollFade() {
}

ScrollFade::~ScrollFade() {
}
void ScrollFade::fade(BITMAP* src, BITMAP* dst, double percent) {
    int x = dst->w * percent;
    int w = dst->w - w;

    rectfill(dst, 0, 0, w, dst->h, 0);
    blit(src, dst, 0, 0, x, 0, w, src->h);
}
```

Derzeit wird das Bild immer von der linken Seite zur rechten geschoben. Sie können die Methoden aber sehr einfach anpassen, um beliebige Richtungen zu unterstützen. Übergeben Sie hierzu im Constructor der Klasse einfach –1, wenn das Bild entgegen der Achsenrichtung verschoben werden soll, +1, wenn es entlang der Achsenrichtung verschoben werden soll oder 0, wenn es entlang dieser Achse überhaupt nicht verschoben werden soll. Beachten Sie jedoch, dass sich das Bild nicht bewegt, falls für beide Achsen 0 übergeben wird.

DoorFade

Beim `DoorFade` gleitet das Bild nach beiden Seiten auseinander, als ob eine Schiebetür geöffnet wird.

Im Grunde genommen wird hier das Bild in 2 Hälften geteilt, die nach links und rechts aus dem Bild gezogen werden. Es wäre also durchaus möglich gewesen, zwei Subbitmaps anzulegen und dann diese entsprechend zu verschieben.

Allerdings wurde aufgrund der Tatsache, dass die hier verwendeten Methoden zustandslos sein sollen (d.h. sie dürfen keine Informationen zwischen den Aufrufen speichern) und das Erzeugen von Subbitmaps eine verhältnismäßig langsame Aktion ist, auf eine solche Lösung verzichtet.

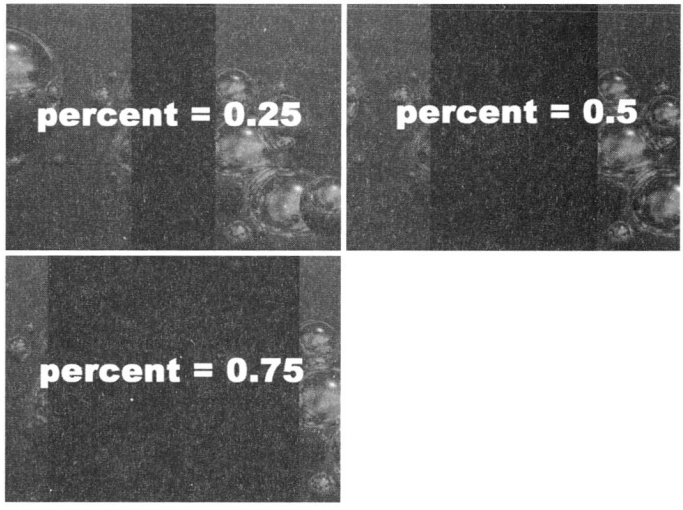


Abbildung 33.7: Ein DoorFade – das Bildet gleitet seitlich auseinander

Stattdessen werden die entsprechenden Bereich, der Bitmap berechnet und dann mittels zweier Aufrufe von `blit()` angezeigt.

```
// ----------------------------------------------------
// -- DoorFade
// ----------------------------------------------------
DoorFade::DoorFade() {
}

DoorFade::~DoorFade() {
}
void DoorFade::fade(BITMAP* src, BITMAP* dst, double percent) {
    int x  = dst->w * percent * 0.5;
    int w  = dst->w/2 - x;
    int x2 = src->w / 2;

    blit(src, dst,  x, 0, 0, 0, w, src->h);
    blit(src, dst, x2, 0, dst->w/2+x, 0, w, src->h);
    if (x >0) {
        rectfill(dst, dst->w/2-x, 0, dst->w/2 +x, dst->h,0);
    }
}
```

BlindFade

Beim `BlindFade` wird grob der Effekt einer Jalousie simuliert. Dies wird erreicht, indem in regelmäßigen Abständen schwarze Balken eingezeichnet werden, die im Laufe des Fades an Höhe gewinnen bis der Schirm komplett schwarz ist.

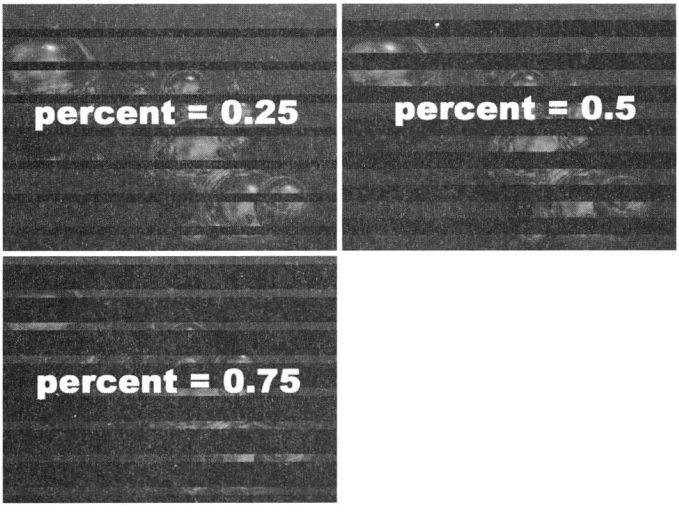


Abbildung 33.8: Der BlindFade

Auch hier ist der eigentliche Code sehr simpel. Es wird die aktuelle Höhe der Balken berechnet, dann werden die Balken mit der entsprechenden Höhe innerhalb einer Schleife gemalt.

```
// ----------------------------------------------------
// -- BlindsFade
// ----------------------------------------------------
BlindsFade::BlindsFade() {
}

BlindsFade::~BlindsFade() {
}
void BlindsFade::fade(BITMAP* src, BITMAP* dst, double percent) {
    int h    = (dst->w / 10);
    int curH = (int) (h * percent);
```

```
    int y    = h;
    blit(src, dst, 0, 0, 0, 0, src->w, src->h);
    for (int a=0; a < 9; ++a) {
        rectfill(dst, 0, y-curH, dst->w, y, 0);
        y += h;
    }
}
```

Testprogramm

Zum Testen wird ein Array mit den einzelnen Fade-Klassen angelegt, dann wird zwischen zwei Bildern mit den entsprechenden Klassen überblendet.

Die fadecol.cpp Datei und ihr Header fadecol.h enthalten die bisher besprochenen Überblendfunktionen. Aus diesem Grund wird auf ihre Abbildung hier verzichtet.

```
#include <allegro.h>
#include "fade.h"
#include "fadecol.h"

BITMAP *doubleBuffer;
volatile int timerCounter;
void timerFunction(void) {
    timerCounter++;
}
END_OF_FUNCTION(timerFunction);

int setGfxMode(int w, int h, int win) {
    int mode = win ? GFX_AUTODETECT_WINDOWED :
GFX_AUTODETECT_FULLSCREEN;
    int colorDepth[] = {16, 15, 32, 24, 0};
    int a = 0;
    while (colorDepth[a]) {
        set_color_depth(colorDepth[a]);
        if (set_gfx_mode(mode, w, h, 0, 0) >= 0) {
            return TRUE;
        }
        ++a;
    }
    return FALSE;
}

BITMAP *createDoubleBuffer() {
    BITMAP *bmp= create_bitmap(SCREEN_W, SCREEN_H);
```

```
    return bmp;
}

int init(int preferWin) {
    allegro_init();

    /* set 640x480 hicolor preferredMode*/
    if (!setGfxMode(640, 480, preferWin)) {
        /* or non preferred mode */
        if (!setGfxMode(640, 480, !preferWin)) {
            return FALSE;
        }
    }

    /* init subsystems */
    install_keyboard();
    install_joystick(JOY_TYPE_AUTODETECT);
    install_sound(DIGI_AUTODETECT, MIDI_AUTODETECT, NULL);
    install_timer();

    /* lock timer vars / functions */
    LOCK_VARIABLE(timerCounter);
    LOCK_FUNCTION(timerFunction);

    /* install logic timer */
    install_int_ex(timerFunction, BPS_TO_TIMER(60));

    doubleBuffer = createDoubleBuffer();
}

void done() {
    destroy_bitmap(doubleBuffer);
}


void fade(BITMAP * from, BITMAP *to, Fade *fade, int ticks) {
    double percent;
    int targetTime   = timerCounter + ticks;
    int curTime      = timerCounter;
    int needsRefresh = FALSE;

    while (timerCounter < targetTime && !key[KEY_ESC]) {
        if (curTime <=timerCounter) {
            percent = (double) (ticks - (targetTime - curTime)) /
           (double) ticks;
```

```
            curTime      = timerCounter+1;
            needsRefresh = TRUE;
        }
        if (needsRefresh) {
            needsRefresh = FALSE;
            if (percent <= 0.5) {
                fade->fade(from, doubleBuffer, percent*2.0);
            } else {
                fade->fade(to, doubleBuffer, 1.0 - (percent-
               0.5)*2.0);
            }
            blit(doubleBuffer, screen, 0, 0, 0, 0, doubleBuffer->w,
           doubleBuffer->h);
        }
    }
}

int main(int argc, char** argv) {
    init(argc > 1);

    srand(time(NULL));

    BITMAP *one, *two;
    one = load_bitmap("image1.tga", NULL);
    two = load_bitmap("image2.tga", NULL);

    const int COUNT = 8;
    Fade *fader[COUNT] = {
        new StripeFade(10),
        new StretchFade(),
        new StripeFade(40),
        new ScrollFade(),
        new SymmetricStripeFade(20),
        new DoorFade(),
        new SymmetricStripeFade(4),
        new BlindsFade(),
    };

    int a = 0;
    while (!key[KEY_ESC]) {
        fade(one, two, fader[a], 90);
        ++a;
        a%=COUNT;
        fade(two, one, fader[a], 90);
        ++a;
        a%=COUNT;
    }
```

```
    for (int a=0; a < COUNT; a++) {
        delete fader[a];
    }
    done();
}
END_OF_MAIN()
```

Crossfades

Das Überblenden auf beziehungsweise von einem schwarzen Bildschirm sollte nun kein Problem mehr sein. Mit diesem Wissen können wir uns nun an die Implementierung von direkten Übergängen zwischen zwei Bildern machen.

Der offensichtlichste Unterschied bei den benötigten Funktionen ist sicherlich, dass wir zwei Ausgangsbilder haben, deren Mischung auf die Zielbitmap gerendert wird.

```
crossfade(BITMAP* img1, BITMAP* img2, BITMAP* dst, double percent);
```

Nach Aufruf der Methode `crossfade()` sollte die Zielbitmap (dst) im angegebenen Verhältnis (`percent`) aus `img1` und `img2` bestehen. Soll zum Beispiel das zweite Bild schrittweise sichtbar werden, so könnte die entsprechende Methode so aussehen:

```
crossfade(BITMAP* img1, BITMAP* img2, BITMAP* dst, double percent) {
    blit(img1, dst, 0, 0, 0, 0, img1->w, img1->h);
    set_trans_blender(0,0,0, (int)(255 * percent));
    draw_trans_sprite(dst, img2, 0, 0);
}
```

Auf die ursprüngliche Bitmap (`img1`) wird das zweite Bild (`img2`) mit einem stetig geringeren Transparenzwert gezeichnet, bis schließlich nur noch das Zielbild zu sehen ist. Die Transparenz eines Bildes bezeichnet man auch als dessen Alphawert. Da hier zwei Bilder durch Veränderung des Alphawertes überblendet werden, spricht man hier auch von einem Alphablending.

Kapseln wir nun diese Funktionalität wieder in einer Klasse, um die Crossfades beliebig austauschen zu können. Hier die Headerdatei für die abstrakte Basisklasse `CrossFade`:

```
#ifndef CROSSFADE_HEADER
#define CROSSFADE_HEADER
```

```
#include <allegro.h>

class CrossFade {
public:
    CrossFade();
    virtual ~CrossFade();
    virtual void fade(BITMAP* img1, BITMAP* img2, BITMAP* dst, double
    percent) = 0;
};

#endif
```

Und hier die Implementierung, die wie schon bei der Fade-Klasse sehr spärlich ausfällt:

```
#include "crossfade.h"

CrossFade::CrossFade() {
}

CrossFade::~CrossFade() {
}
```

Die Methode zum Testen dieser Funktionalität ist sogar noch einfach zu Implementieren als bei den Ein-/Ausblendroutinen.

```
void crossfade(BITMAP * from, BITMAP *to, CrossFade *crossfade, int
ticks) {
    double percent;
    int targetTime   = timerCounter + ticks;
    int curTime      = timerCounter;
    int needsRefresh = FALSE;

    while (timerCounter < targetTime && !key[KEY_ESC]) {
        if (curTime <=timerCounter) {
            percent = (double) (ticks - (targetTime - curTime)) /
           (double) ticks;

            curTime      = timerCounter+1;
            needsRefresh = TRUE;
        }
        if (needsRefresh) {
            needsRefresh = FALSE;
            crossfade(from, to, doubleBuffer, percent);
            blit(doubleBuffer, screen, 0, 0, 0, 0, doubleBuffer->w,
           doubleBuffer->h);
        }
    }
}
```

Wenn Sie diese Funktion mit der entsprechenden Funktion für das Ein-/Ausblenden (`fade()`) vergleichen, so fällt Ihnen sicherlich auf, dass nun die Prozentzahl direkt verwendet werden kann.

Alphablending

Wie bereits oben erwähnt, wird beim Alphablending das zweite Bild über dem ersten eingeblendet.

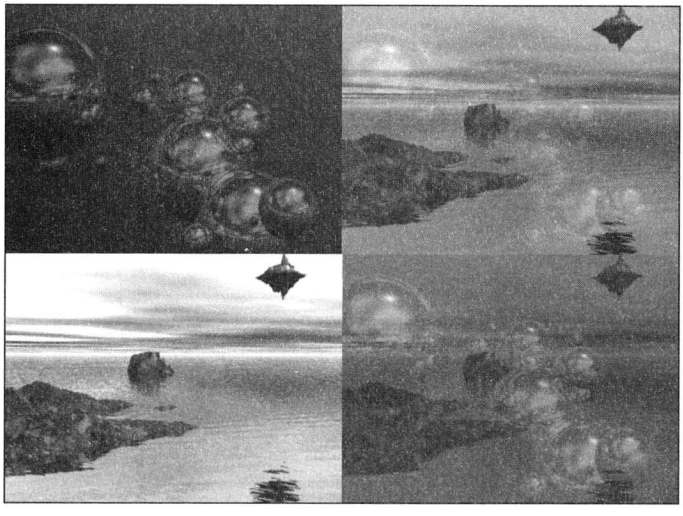


Abbildung 33.9: Im Uhrzeigersinn vom Ausgangsbild zum Endbild

Der Quellcode für diese Routine entspricht dem oben bereits abgebildeten Code, nur diesmal ist er in der `CrossFade`-Klasse gekapselt:

```
AlphaBlending::AlphaBlending() {
}

AlphaBlending::~AlphaBlending() {
}

void AlphaBlending::crossfade(BITMAP* img1, BITMAP* img2, BITMAP*
dst, double percent) {
```

```
    blit(img1, dst, 0, 0, 0, 0, img1->w, img1->h);
    set_trans_blender(0, 0, 0, (int) (255 * percent));
    draw_trans_sprite(dst, img2, 0, 0);
}
```

Mosaic

Der Mosaic Crossfade ist auch einer der Effekte aus der Kategorie: wenig Aufwand, große Wirkung.

Die Idee ist recht einfach: Es wird angenommen, dass der Bildschirm zu Beginn nur aus 2 Reihen mit jeweils 2 Pixel besteht. Diese 4 Pixel füllen den gesamten Bildschirm. Im Laufe der Zeit erhöht sich die Auflösung des Bildschirms, und das Bild wird immer feinkörniger und damit auch besser zu erkennen.

Um dies in einem Überblendeffekt zu nutzen, wird zusätzlich noch ein Alphablending benutzt. Erst ist das Bild also grobpixelig und durchsichtig und am Ende dann fein aufgelöst und undurchsichtig.

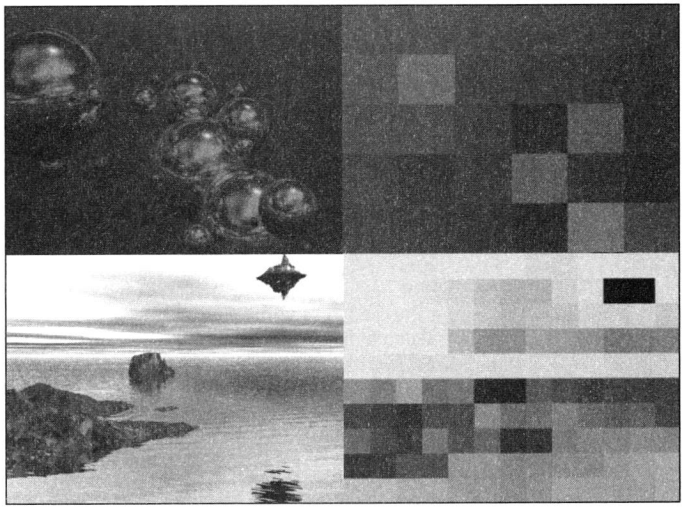


Abbildung 33.10: Der Mosaic Crossfade

Es gibt eine einfache Methode, um ein Bild pixelig darzustellen:

- ✔ Zuerst wird das Bild verkleinert,
- ✔ Dann wieder auf die Originalgröße gebracht.

Je nachdem wie stark das Bild im ersten Schritt verkleinert wurde, umso stärker der Pixeleffekt. Nun stellt uns dies aber vor ein kleines Problem: Um sicherzustellen, dass keinerlei Bilddaten überschrieben werden, brauchen wir einen weiteren Puffer. Dieser dient quasi als Zwischenablage für die kleinere Variante des Bildes.

```
void MosaicCrossFade::mosaicBlit(BITMAP *dest, BITMAP *source,
unsigned int blockSize) {
    int dw = source->w;
    int dh = source->h;
    if (blockSize > 0) {
        dw /= blockSize;
        dh /= blockSize;
    }

    stretch_blit(source, fxBuffer,
                 0, 0, source->w, source->h,
                 0, 0, dw, dh);

    stretch_blit(fxBuffer, dest,
        0, 0, dw, dh,
        0, 0, source->w, source->h);
}
```

Der `fxBuffer` ist der oben erwähnte Pufferspeicher. Die entsprechende Bitmap muss im Konstruktor der `MosaicCrossFade` erzeugt werden. Nun wollen wir aber die zweite Bitmap nicht nur gepixelt, sondern auch transparent anzeigen. Da es in Allegro keine `draw_trans_stretched_sprite()`-Funktion gibt, müssen wir auch hier wieder auf einen Puffer ausweichen. Zuerst malen wir die gepixelte Variante des zweiten Bildes in die Bufferbitmap, dann sorgen wir dafür, dass sie transparent angezeigt wird:

```
void MosaicCrossFade::crossfade(BITMAP* img1, BITMAP* img2, BITMAP*
dst, double percent) {
    int bs1 = MID(1, (int) (img1->h * (1-percent)), 640);
    int bs2 = MID(1, (int) (img2->h * percent), 640);

    mosaicBlit(dst, img1, bs2);
    mosaicBlit(transBuffer, img2, bs1);
    set_trans_blender(0, 0, 0, (int)(255*percent));
```

```
    draw_trans_sprite(dst, transBuffer, 0, 0);
}
```

Nachdem nun die »Arbeitstiere« dieser Klasse ihre Geheimnisse bloßgelegt haben, folgen hier noch die Konstruktoren und der Destruktor:

```
MosaicCrossFade::MosaicCrossFade() : w(0), h(0), fxBuffer(NULL),
transBuffer(NULL) {
}

MosaicCrossFade::MosaicCrossFade(int w, int h) {
    this->w = w;
    this->h = h;
    fxBuffer    = create_bitmap(w, h);
    transBuffer = create_bitmap(w, h);
}


MosaicCrossFade::~MosaicCrossFade() {
    if (fxBuffer) {
        destroy_bitmap(fxBuffer);
    }
    if (transBuffer) {
        destroy_bitmap(transBuffer);
    }
}
```

Wie gesagt, der Code ist recht einfach, doch der Effekt ist sehenswert.

DoorCrossFade

Diese Überblendung ist eine Variante der vorher besprochenen DoorFades. Das erste Bild wird nach links und rechts aus dem sichtbaren Bereich geschoben, im größer werdenden Bereich in der Mitte wird das neue Bild angezeigt. Sie können dieses Beispiel als Anregung nutzen, um auch die anderen Fades in Crossfades umzuwandeln.

Der Code entspricht weitestgehend dem der `DoorFade` Routine, nur wird statt einem schwarzen Hintergrund das zweite Bild gezeigt.

```
void DoorCrossFade::crossfade(BITMAP* img1, BITMAP* img2, BITMAP*
dst, double percent) {
    int gap  = (int) (dst->w * percent);
    int x    = (int) (gap * 0.5);
    int w    = (int) (dst->w/2 - x);
```

```
    int x2   = (int) (img1->w / 2);

    blit(img2, dst,  w, 0, w, 0, gap, img2->h);

    blit(img1, dst,  x, 0, 0, 0, w, img1->h);
    blit(img1, dst, x2, 0, dst->w/2+x, 0, w, img1->h);

}
```

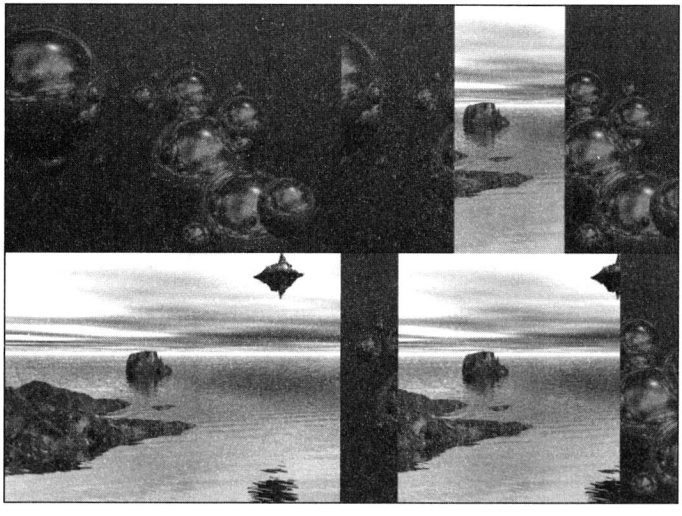


Abbildung 33.11: DoorCrossFade

Sie können die Methoden kombinieren, um weitere Effekte zu erzielen.

DoorZoomCrossFade

Bei diesem Überblendeffekt handelt es sich um eine Kombination zweier Techniken. Das erste Bild wird aus dem Bild geschoben, das zweite Bild wird im Hintergrund immer größer.

Auch hier ist der Quellcode den bekannten `DoorFade` Routinen wieder sehr ähnlich.

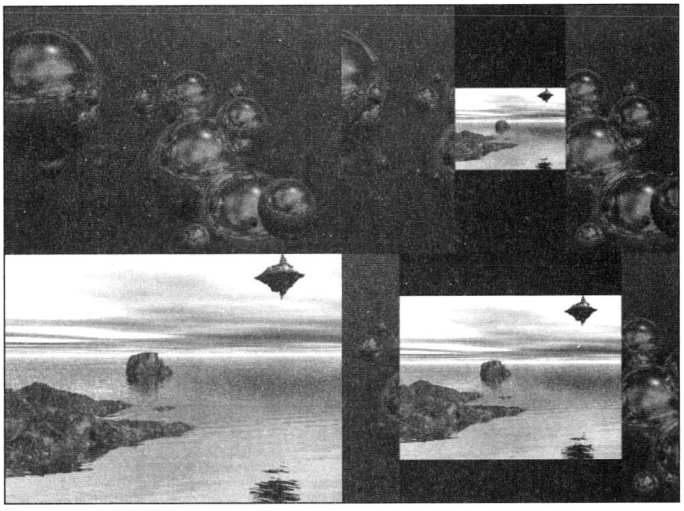


Abbildung 33.12: DoorZoomCrossFade

```
void DoorZoomCrossFade::crossfade(BITMAP* img1, BITMAP* img2, BITMAP*
dst, double percent) {

    int gap  = (int) (dst->w * percent);
    int x    = (int) (gap * 0.5);
    int w    = (int) (dst->w/2 - x);
    int x2   = (int) (img1->w / 2);

    // Berechnen der Größe des Bildes
    int dw = (int) (img2->w * percent);
    int dh = (int) (img2->h * percent);
    int dx = (dst->w - dw) / 2;
    int dy = (dst->h - dh) / 2;

    if (x > 0) {
        // Bereich löschen
        rectfill(dst, dst->w/2-x, 0, dst->w/2 +x, dst->h,0);
        // Anzeige in der berechneten Größe
        stretch_blit(img2, dst,  0, 0, img2->w, img2->h, dx, dy, dw,
       dh);
    }
```

```
    blit(img1, dst,  x, 0, 0, 0, w, img1->h);
    blit(img1, dst, x2, 0, dst->w/2+x, 0, w, img1->h);

}
```

Die DoorFade-Routinen bieten noch viel Spielraum für eigene Ideen. Experimentieren Sie ruhig etwas, und versuchen Sie die Effekte zu kombinieren.

RollOut

Bei diesem Crossfade wird das Bild von oben nach unten zusammengerollt.

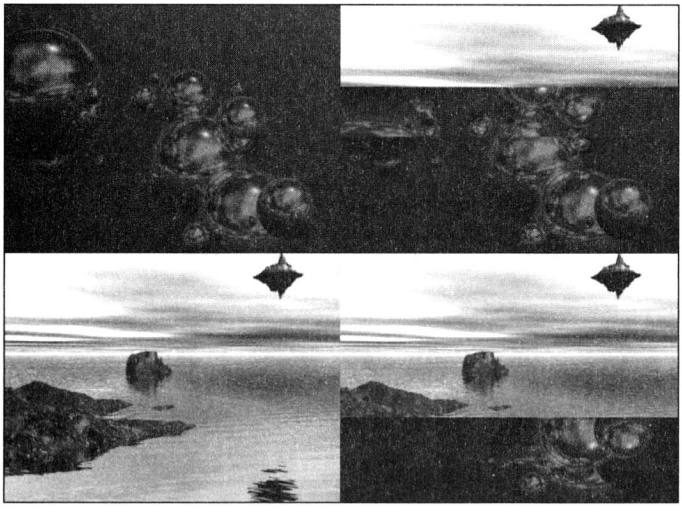


Abbildung 33.13: RollOut – Das Bild wird zusammengerollt

Stellen Sie sich vor, Sie legen zwei Plakate genau aufeinander. Nun rollen Sie das obere Plakat zusammen, wodurch das untere Bild sichtbar wird.

Diesen Effekt kann man mit minimalem Aufwand simulieren. Der Falz bewegt sich von oben nach unten. Oberhalb des Falzes wird das zweite Bild sichtbar. In einem Bereich, welcher dem Durchmesser der Rolle ent-

spricht, wird ein Teil des ersten Bildes gespiegelt angezeigt (siehe auch Abbildung 33.14)

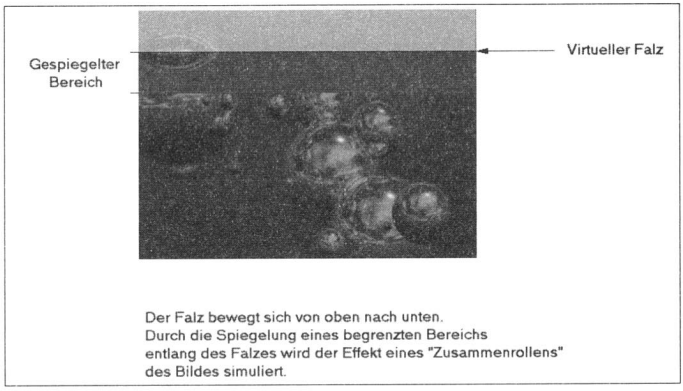


Abbildung 33.14: RollOut-Diagramm

Diese Methode ist nicht 100% korrekt. Wir vernachlässigen die perspektivische Verzerrung aufgrund der Wölbung des eingerollten Bereiches. Allerdings fällt dies in der Bewegung nicht auf.

```
void RollOut::crossfade(BITMAP* img1, BITMAP* img2, BITMAP* dst,
double percent) {

    int y = (int) (percent * img1->h);
    int h = MIN(y, img1->h * 0.2);

    blit(img2, dst, 0, 0, 0, 0, img2->w, y);
    blit(img1, dst, 0, y+h, 0, y+h, img1->w, img1->h - y - h);

    // Spiegel des eingerollten Bereichs
    for (int a=0; a < h; a++) {
        blit(img1, dst, 0, y+a, 0, y+h-a, img1->w, 1);
    }
}
```

Der eingerollte Bereich wird manuell gespiegelt, da die Allegro-Funktionen nur das Spiegeln der gesamten Bitmap vorsehen, nicht aber das Spiegeln eines bestimmten Bereiches.

Aber wie sie sehen können, kann man dieses Problem durch eine einfache Schleife lösen, in der die einzelnen Zeilen in umgekehrter Reihenfolge angezeigt werden.

Auch diese Überblendung eignet sich sehr gut für Variationen. Anstatt ein Bild zusammenzurollen, könnten Sie auch das zweite Bild über dem ersten entrollen. Oder Sie rollen das Bild von beiden Seiten gleichzeitig zusammen.

Weitere Effekte

Die Liste möglicher Crossblend-Routinen ist nahezu endlos. Wenn Sie weitere Ideen brauchen, dann werfen Sie einen Blick auf die Effekte, die Videoschnittsoftware anbieten. Achten bei Ihrem nächsten Kinobesuch auf die Überblendeffekte oder fangen Sie einfach an wild Effekte zu kombinieren.

34 Wegfindung mit A*

Den schnellsten Weg von A nach B zu finden, ist gar nicht so einfach. Leider wird aber genau dies sehr häufig in Spielen benötigt. In diesem Kapitel lernen Sie den A*-Algorithmus (A-Stern- oder A-Star-Algorithmus) anzuwenden, der immer den kürzesten Weg zum Ziel findet – wenn es denn einen gibt.

Wohin des Weges

Wegfindungsalgorithmen sind ein wichtiger Bestandteil von vielen Spielen. In Abenteuerspielen werden sie benutzt, damit der Held den Weg durch das Zimmer findet und nicht am Tisch hängen bleibt. In Strategiespielen werden sie verwendet, damit die Einheiten den richtigen Weg finden. In Rollenspielen werden Sie benutzt, damit die Gegner den Weg zum Helden finden.

Der A* (sprich: »A Star« oder »A Stern«)-Algorithmus ist einer der am häufigsten verwendeten Algorithmen, um den kürzesten Weg zwischen zwei Punkten zu finden.

Die Idee hinter A*

Die Idee hinter dem A*-Algorithmus ist, die Anzahl der möglichen Wege dadurch zu reduzieren, dass man immer dem aktuell besten Weg zuerst nachgeht.

Vom Ausgangspunkt werden erst alle möglichen Nachbarpunkte ermittelt. In einer Tile Map sind dies die 8 umliegenden Tiles. Allerdings ist der A*-Algorithmus nicht auf Kacheln beschränkt. Wenn man die möglichen Positionen und ihre Verbindungen grafisch darstellt, erhält man ein Netz, bei dem die Knoten die begehbaren Felder repräsentieren.

Der A* findet nun den kürzesten Weg vom Ausgangsknoten (Start Node) zum Zielknoten (Target Node).

Ein wichtiger Teil des Algorithmus ist die Funktion, die den verbleibenden Weg von einem Knoten zum Zielknoten abschätzt. Häufig wird einfach der geometrische Abstand der Punkte genommen, in anderen Fällen

müssen noch weitere Faktoren in Betracht gezogen werden, wie zum Beispiel die Beschaffenheit des Untergrunds in einem Strategiespiel. Man spricht hier von den Kosten eines Weges zwischen zwei Knoten.

Vom Startknoten aus durchsucht nun der Algorithmus alle Felder, bis er den Zielknoten gefunden hat. Dabei werden die geschätzten Kosten der Knoten verglichen und immer zuerst der günstigste Weg weiterverfolgt.

Hierzu werden zwei Listen benutzt. In der einen Liste werden alle Knoten abgespeichert, die noch untersucht werden müssen. In der zweiten Liste stehen die Knoten, die bereits besucht wurden.

Der Algorithmus nimmt nun immer den besten Knoten aus der Liste der offenen Knoten. Ist dieser Knoten das Ziel, dann ist die Suche erfolgreich abgeschlossen. Ist dieser Knoten nicht das Ziel, dann werden die Nachfolger dieses Knotens zur Liste der zu untersuchenden Knoten hinzugefügt und der aktuelle Knoten in die Liste der bereits besuchten Knoten übernommen.

Nun beginnt das Spiel von neuem, bis der Zielknoten gefunden wird.

Pseudocode

Der Vorgang wird klarer, wenn Sie einen Blick auf den zugehörigen Pseudocode werfen.

```
Start Knoten erzeugen
End Knoten erzeugen
Start Knoten in die Liste der offenen Knoten übernehmen
Solange es offene Knoten gibt
    Nenne den Knoten mit den geringsten Kosten Node
    Ist Node == Start?
        Ja: Ziel Erreicht, Programm beenden

    Erzeuge alle möglichen Nachfolger von Node
    Für jeden Nachfolger
       Setze die Kosten des aktuellen Nachfolgers
       Finde den Nachfolger in der Liste offener
                                              Knoten
       Ist der gefundene Knoten billiger?
           Ja: verwerfe den aktuellen Nachfolger
               und gehe zum nächsten

       Suche den Knoten in der Liste bereits besuchter
       Knoten
```

```
            Ist der gefundene Knoten billiger?
                Ja: verwerfe den aktuellen Nachfolger
                    und gehe zum nächsten

            Wenn die Listen keinen billigeren Knoten
            Enthalten, dann
                Entferne die teureren Versionen von den
                Listen
                Füge den Nachfolger zur Liste offener Knoten
                hinzu
        Ende der Nachfolger Schleife
        Entferne Node von der Liste offener Knoten
        Füge Node der Liste geschlossener Knoten hinzu
    Ende Schleife aller offener Knoten
```

Kosten

Die »Kosten« eines Knotens spielen bei der Entscheidung, welcher Knoten als nächster besucht wird, eine entscheidende Rolle. Die einfachste Lösung ist, die geometrische Entfernung zu benutzen. Diese berechnet sich aus der Wurzel des Quadrates der Summe der Entfernungen auf der X- und Y-Achse:

```
dist = sqrt( dx*dx + dy *dy);
```

Allerdings ist die Wurzelberechnung eine sehr zeitintensive Rechnung. Glücklicherweise können wir die Wurzel komplett vermeiden, da wir ja nicht die wirkliche Entfernung brauchen, sondern nur die Kosten der einzelnen Knoten vergleichen müssen.

Das bedeutet: Solange die Kosten für einen weiter entfernten Knoten größer sind als die Kosten für einen näheren Knoten ist es egal, wie die Kosten berechnet werden. Es ist also absolut legitim, das Quadrat der Summen zu benutzen anstatt die Wurzel.

A* beobachten

Das Beispielprogramm zum A-Stern-Algorithmus sucht einen Weg in einem einfachen Level. Bei jedem Druck der [Leer]-Taste wird der nächste Schritt der Suche durchgeführt. Dies erlaubt es Ihnen, den Suchvorgang Schritt für Schritt zu beobachten.

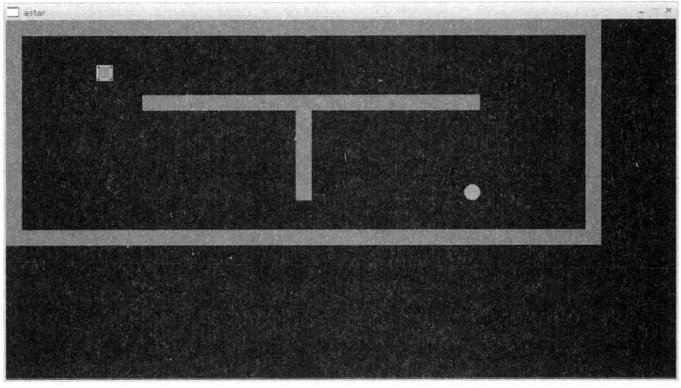


Abbildung 34.1: Die Ausgangssituation

Zu Beginn ist nur der Startknoten (oben links) in der Liste der offenen Knoten. Das Ziel ist der türkise Kreis in der rechten unteren Ecke.

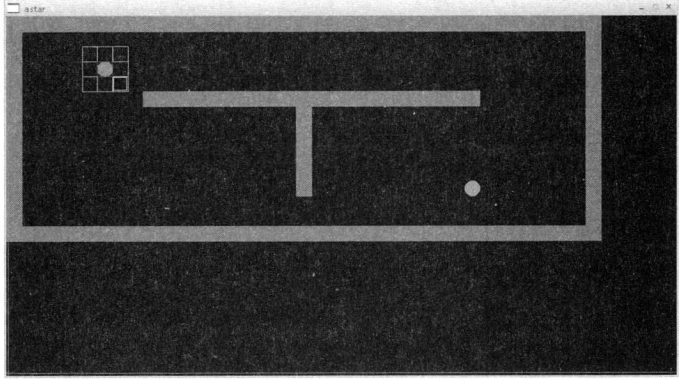


Abbildung 34.2: Der erste Schritt der Suche

Im ersten Schritt werden alle umliegenden Felder überprüft. Das Feld, das dem Ziel am nächsten ist, wird bevorzugt.

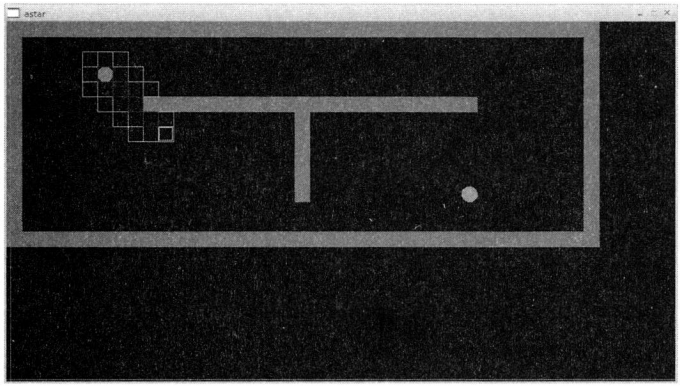


Abbildung 34.3: Auf dem Weg

Der Start der Suche ist kein Problem für den A*-Algorithmus – er marschiert direkt auf das Ziel zu.

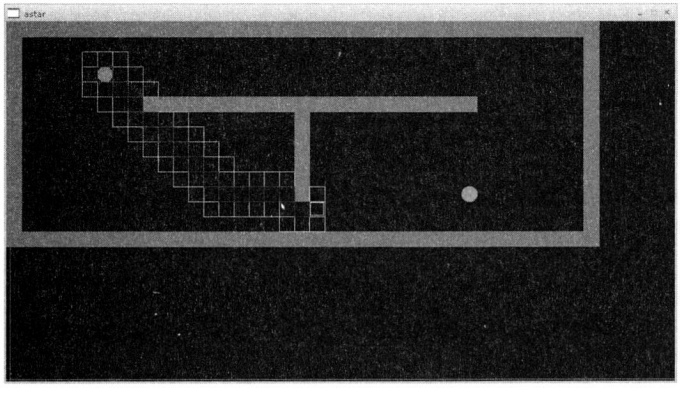


Abbildung 34.4: Ein erstes Hindernis

Am Fuße des Ts kommt es zur ersten Unterbrechung der direkten Linie. Aber auch hier ist schnell eine Lösung gefunden.

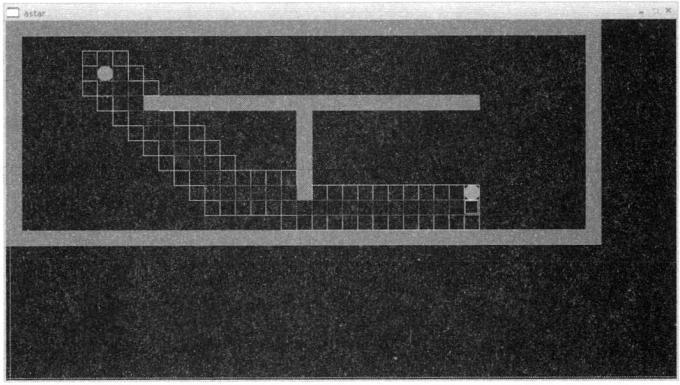


Abbildung 34.5: Fast da

Beachten Sie, wie schnell der Algorithmus den Weg findet. Innerhalb kürzester Zeit steht er direkt vor dem Ziel.

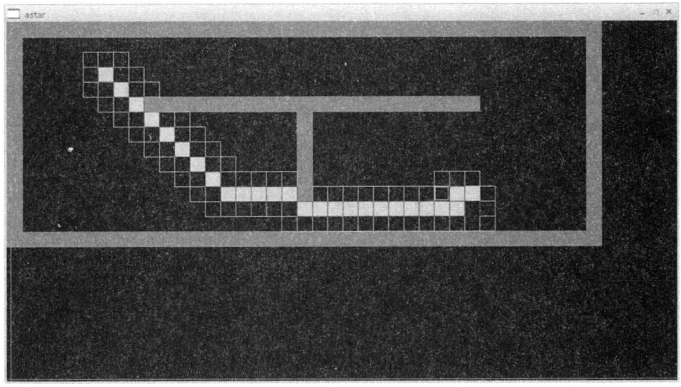


Abbildung 34.6: Ziel gefunden

Sobald das Ziel erreicht ist, wird vom Ziel aus rückwärts gegangen, um den Pfad zu finden. In diesem Fall war die Suche sehr schnell erfolgreich. Wenn wir es dem Algorithmus etwas schwerer machen, dann braucht er ein paar Versuche mehr, bis das Ziel erreicht ist.

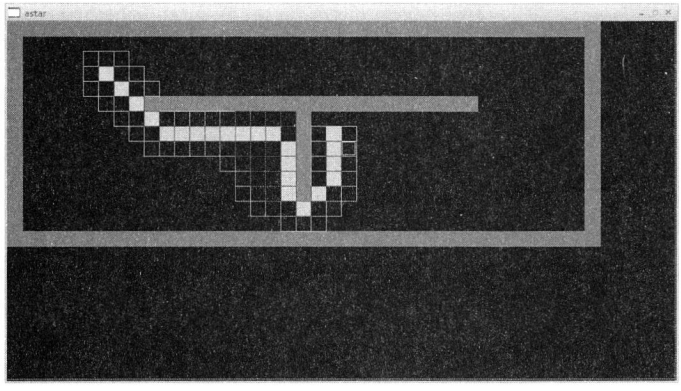


Abbildung 34.7: Ein etwas schwererer Weg

Aber auch hier ist die Lösung schnell gefunden.

Wenn wir es dem Algorithmus etwas schwerer machen, indem wir das T bis zum Boden verlängern, und dadurch eine Sackgasse schaffen, dann braucht er auch ein paar Iterationen länger, um einen Weg zu finden.

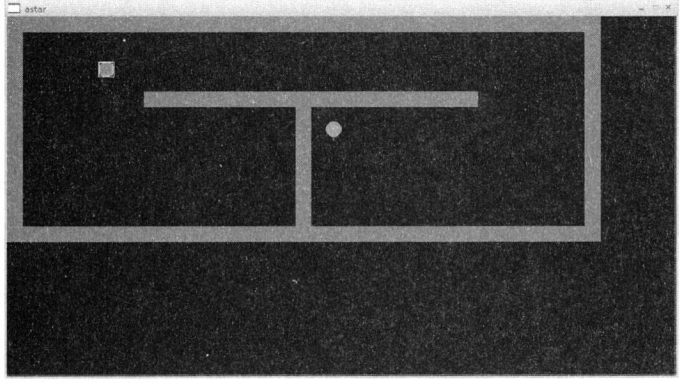


Abbildung 34.8: Karte mit Sackgasse

Das Problem hier ist auch, dass das Ziel sehr nahe am Querbalken liegt, dadurch sind viele falsche Positionen sehr nahe am Ziel.

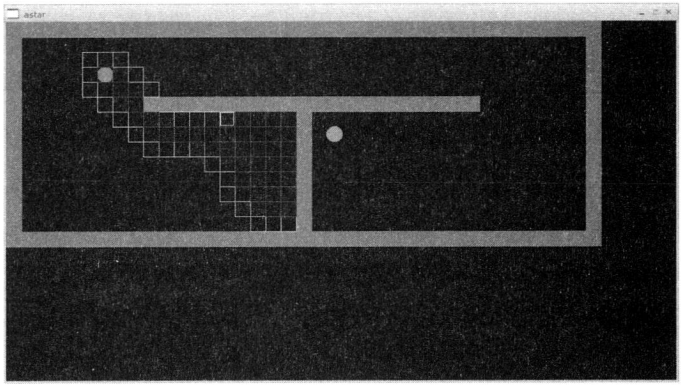


Abbildung 34.9: In der Sackgasse

Der Algorithmus sucht recht lange in diesem Fall, bevor er merkt, dass der Weg nach unten eine Sackgasse ist.

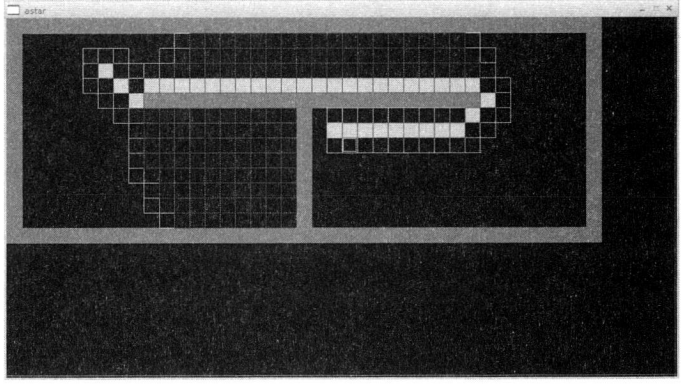


Abbildung 34.10: Geschafft!

Dieses Mal braucht er einige Durchgänge, bevor es zum Ziel schafft – und der Weg ist in diesem Fall aus unserer Sicht aus nicht einmal optimal (rein von den Zahlenwerten her ist der Weg optimal).

Implementierung

Nun aber zur Umsetzung. Die Karte wird wieder als String im Programm gespeichert. Dadurch können Änderungen schnell vorgenommen werden.

```
char *map =
"#######################################"
"#                                     #"
"#                                     #"
"#     A                               #"
"#                                     #"
"#        #######################      #"
"#                 #                   #"
"#                 # B                 #"
"#                 #                   #"
"#                 #                   #"
"#                 #                   #"
"#                 #                   #"
"#                 #                   #"
"#                 #                   #"
"#######################################";
```

Nicht begehbare Felder sind mit einem `#` gekennzeichnet. Der Startpunkt ist durch ein `A` und das Ziel durch ein `B` markiert.

Alle Klassen verzichten auf das Verstecken ihrer Interna, damit die Arbeitsweise besser dargestellt werden kann. In ihrer endgültigen Version sollten Sie die Interna natürlich als `private` deklarieren.

Der Knoten

Jeder Knoten in der Karte besitzt neben seiner Position 3 Werte, die seine Kosten bestimmen.

Variable	Bedeutung
f	Gesamtkosten
g	Kosten den Punkt zu erreichen
h	Geschätzte Kosten von dem Knoten zum Ziel

Tabelle 34.1: Variablen für die Kosten eines Knotens

Zusätzlich speichert jeder Knoten auch den Knoten, von dem aus er erreicht wurde. Dies erlaubt es uns am Ende den Weg zurückzuverfolgen.

```
struct Node {
    int x, y;
    int f,g,h;

    Node *prev;

    Node(int x=-1, int y=-1) {
        this->x = x;
        this->y = y;

        g = 0;
        h = 0;
        f = 0;

        prev = NULL;
    }

    int distance(Node *goal) {
        int dx = goal->x - x;
        int dy = goal->y - y;

        return dx*dx + dy*dy;
    }

    bool isEqual(Node *node) {
        return this->x == node->x
               && this->y == node->y;
    }
};
```

Damit wir die Liste der offenen Knoten sortieren können, brauchen wir einen Funktor, der zwei Node Strukturen vergleicht.

```
class CompareNode {
public:
    bool operator() ( const Node *lhs, const Node *rhs ) const {
        return lhs->f > rhs->f;
    }
};
```

Damit wir uns etwas Schreibarbeit ersparen, definieren wir noch ein paar Typen für die Liste, deren Iteratoren und einen Stack von Node-Zeigern.

```
typedef vector<Node*> NodeList;
typedef NodeList::iterator NodeIterator;

typedef stack<Node*> NodeStack;
```

Die Suche

Um eine Suche zu beginnen, wird die Methode `startSearch()` benutzt, die Start- und Zielpunkt als Parameter übernimmt.

```
void startSearch(Node *start, Node *end) {
    // Sicherstellen, dass die Listen leer sind
    for (unsigned int a=0; a < open.size(); ++a) {
        delete open[a];
    }
    for (unsigned int a=0; a < closed.size(); ++a) {
        delete closed[a];
    }

    open.clear();
    closed.clear();

    this->start = start;
    this->end   = end;

    start->h = start->distance(end);
    start->f = start->h;

    count = 0;

    // Die open liste ist ein Heap
    // d.h. Sie ist sortiert, und hat
```

```
    open.push_back(start);
    push_heap( open.begin(),
               open.end(),
               CompareNode() );
}
```

Das eigentliche Suchen übernimmt die `step()`-Methode, die den oben beschriebenen Algorithmus implementiert.

```
bool step() {
    if (open.empty()) {
        return false;
    }

    Node *n = open.front();
    pop_heap(open.begin(), open.end(), CompareNode());
    open.pop_back();


    int nx, ny;

    NodeStack stack;
    // Am Ziel angekommen?
    if (n->isEqual(end)) {
        // Im Ziel knoten den Weg setzten
        end->prev = n->prev;
        return true;
    } else {
        // Alle umliegenden Felder
        for (int y=-1; y <= 1; ++y) {
            for (int x=-1; x<= 1; ++x) {
                if (x != 0 || y != 0) {
                    nx = x + n->x;
                    ny = y + n->y;

                    if (!n->prev ||
                        (n->prev->x != nx  ||
                         n->prev->y != ny) &&
                         isWalkable(nx, ny)
                        ) {

                        Node *node = new Node(nx, ny);
                        stack.push(node);
                    }
                }
```

```
            }
        }
    }

    while (!stack.empty()) {
        Node *node = stack.top();
        stack.pop();
        int newg = n->g + node->distance(n);
        // Knoten in open suchen
        NodeIterator openListItor = findInList(
                                        open, node);
        Node *openListResult      = NULL;

        // gefunden?
        if (openListItor != open.end()) {
            openListResult = *openListItor;
            if (openListResult->g <= newg) {
                // Wir haben bereits einen
                // besseren Weg zu diesem Knoten
                delete node;

                // Zum nächsten Knoten
                continue;
            }
        }

        // Knoten in closed suchen
        NodeIterator closedListItor = findInList(
                                        closed, node);
        Node *closedListResult      = NULL;

        // gefunden?
        if (closedListItor != closed.end()) {
            closedListResult = *closedListItor;
            if (closedListResult->g <= newg) {
                // Wir haben bereits einen
                // besseren Weg zu diesem Knoten
                delete node;

                // Zum nächsten Knoten
                continue;
            }
        }
```

```
            // der neue Knoten ist der bisher beste
            // Weg zu diesem Punkt
            node->prev = n;
            node->g    = newg;
            node->h    = node->distance(end);
            node->f    = node->g + node->h;

            // Den Knoten aus open und closed
            // Liste entfernen (falls er
            // denn dort gefunden wurde)
            if (openListResult) {
                delete openListResult;
                openListResult = NULL;
                open.erase(openListItor);

                // Wenn ein Knoten entfernt wird,
                // dann muss neu sortiert werden.
                make_heap( open.begin(),
                           open.end(),
                           CompareNode());
            }
            if (closedListResult) {
                delete closedListResult;
                closedListResult = NULL;
                closed.erase(closedListItor);
            }

            // Den neuen Knoten hinzufügen
            open.push_back(node);

            // den gerade hinzugefügten Knoten
            // in den Heap integrieren
            push_heap( open.begin(),
                       open.end(),
                       CompareNode());
        }
        closed.push_back(n);
        return false;
    }
```

Diese Methode nutzt einige Hilfsroutinen. Zum einen die Methode `findInList()`, die den Iterator auf einen bestimmten `Node` zurückliefert. Diese Routine ist als lineare Suche des Vectors implementiert.

```
NodeIterator findInList(NodeList &list, Node *n) {
    NodeIterator itor = list.begin();
    while (itor != list.end()) {
        Node *node = *itor;
        if (node->isEqual(n)) {
            return itor;
        }
        ++itor;
    }
    return list.end();
}
```

Eine weitere kleine, aber wichtige Methode ist `isWalkable()`, die überprüft, ob eine bestimmte Position überhaupt passierbar ist.

```
bool isWalkable(int x, int y) {
    return map[x + y *map_w] != '#';
}
```

Das Hauptprogramm

Das Hauptprogramm dient nun nur noch der Anzeige des Suchstatus. Sie können mit der Maus über offene Knoten fahren und sich die Kosten des Knotens anzeigen lassen.

Ein Druck auf die Leer-Taste geht weiter zum nächsten Schritt der Suche.

```
void drawList(NodeList &list, int col, int tw, int th, bool isOpen) {

    if (list.empty()) {
        return;
    }

    NodeIterator itor = list.begin();
    Node *node = *itor;

    rect(doubleBuffer,
           node->x     * tw,  node->y   * th,
           (node->x+1) * tw, (node->y+1)* th,
           col);
    ++itor;

    // Der erste Knoten ist der beste Kandidat
    // (in der open Liste)
    if (isOpen) {
```

```
            rect(doubleBuffer,
                 node->x    * tw+1,  node->y   * th+1,
                (node->x+1) * tw-2, (node->y+1) * th-2,
                 col);
        }

        while (itor != list.end()) {
            node = *itor;
            rect(doubleBuffer,
                     node->x * tw, node->y * th,
                    (node->x+1) * tw, (node->y+1) * th,
                    col);
            ++itor;
        }
    }

    int main(int , char**) {
        init(880, 480, 60, false);

        map_w = strlen(map) / map_h;

        int startX, startY;
        int endX, endY;

        int wallCol   = makecol(128,128,128);
        int startCol  = makecol(0,200,0);
        int endCol    = makecol(0,200,200);
        int openCol   = makecol(100,250,100);
        int closedCol = makecol(200,050,050);

        int tw = 20;
        int th = 20;

        char *pos = map;

        // Start und end suchen
        for (int y=0; y < map_h; ++y) {
            for (int x=0; x < map_w; ++x) {

                switch (*pos) {
                    case 'A':
                        startX = x;
                        startY = y;
                        break;
```

```
                case 'B':
                    endX = x;
                    endY = y;
                    break;
            }

            ++pos;
        }
    }


    Node *start = new Node(startX, startY);
    Node *end   = new Node(endX, endY);

    AStar search(start, end);

    bool spacePressed = false;
    bool found = false;
    while (!key[KEY_ESC]) {
        clear(doubleBuffer);
        char *pos = map;
        for (int y=0; y < map_h; ++y) {
            for (int x=0; x < map_w; ++x) {

                switch (*pos) {
                    case '#':
                        rectfill(doubleBuffer,
                                 x * tw, y*th,
                                (x+1)*tw, (y+1)*th,
                                 wallCol);
                        break;


                    case ' ':
                        break;

                }

                ++pos;
            }
        }

        if (key[KEY_SPACE]) {
            if (!spacePressed) {
                spacePressed = true;
```

```
        if (!found) {
            found = search.step();
        }
    }
} else {
    spacePressed = false;
}

circlefill(doubleBuffer, startX*tw + tw/2,
           startY*th +th/2, tw/2, startCol);
circlefill(doubleBuffer, endX*tw + tw/2,
           endY*th +th/2, tw/2, endCol);

if (found) {

    Node *node = end;
    while (node != NULL) {
        rectfill(doubleBuffer,
                 node->x * tw,
                 node->y * th,
                 (node->x+1) * tw,
                 (node->y+1) * th,
                 openCol);
        node = node->prev;
    }
}

// open list anzeigen
drawList(search.open,   openCol,
         tw, th, true);
drawList(search.closed, closedCol,
         tw, th, false);

int mx = mouse_x;
int my = mouse_y;
vline(doubleBuffer, mx, 0, SCREEN_H, startCol);
hline(doubleBuffer, 0, my, SCREEN_W, startCol);
mx /= tw;
my /= th;
if (mx >=0 && my >=0
    && mx < map_w && my < map_h) {
    Node foo(mx, my);
    NodeIterator curItor =
                 findInList(search.open, &foo);
    if (curItor != search.open.end()) {
```

```
                textprintf_centre(
                    doubleBuffer,
                    font,
                   mouse_x, mouse_y,
                  makecol(255,255,255),
                  "f: %i   g: %i    h:%i" ,
                   (*curItor)->f,
                   (*curItor)->g,
                   (*curItor)->h);
            } else {
                textprintf_centre(
                      doubleBuffer,
                      font,
                      mouse_x, mouse_y,
                      makecol(255,255,255),
                      "Nicht in der Liste" );
            }
        }

        show();
    }

    return 0;
}
END_OF_MAIN()
```

Den kompletten Code finden Sie natürlich auf der beliegenden CD.

35 Scripting mit Lua

Scriptsprachen sind eine gute und schnelle Methode ein Spiel zu erweitern. In diesem Kapitel stelle ich Ihnen Lua vor, eine sehr einfach einzusetzende Scriptsprache, die sich hervorragend für Spiele eignet.

Lua ist eine freie, schnelle und gut zu erweiternde Sprache. Neben der Nutzung als Scriptsprache für andere Applikationen wird Lua auch immer häufiger selbstständig eingesetzt.

Derzeit wird Lua von einigen Herstellern von Spielen benutzt (unter anderem Lucasarts und Bioware), und andere haben Interesse bekundet, Lua in zukünftigen Spielen einzusetzen.

Lua downloaden und installieren

Sie können die aktuelle Version von Lua immer über die Webseite *http://www.lua.org/* beziehen. Da die Sprache häufige Updates erhält, ist dieser Weg der empfehlenswerteste.

Laden Sie die aktuelle Version herunter (derzeit ist Version 5.0 aktuell), und entpacken Sie sie in ein Verzeichnis Ihrer Wahl, zum Beispiel `~/lua-5.0/` (Unix) oder `c:\lua-5.0\` (Windows).

Kompilieren Sie Sourcefiles mit dem enthaltenen makefile, und kopieren Sie dann die `include` und `lib` Dateien in das entsprechende Verzeichnis Ihres Compilers.

Und schon sind Sie einsatzbereit.

Lua-Scripte ausführen

Um in Ihrem Programm Lua Skripte ausführen zu können, bedarf es nicht viel mehr als des Inkludierens der Lua Header und des Aufrufens einiger weniger Funktionen.

Das Kernstück der Lua-Anbindung an Ihre Programme ist der `lua_State`. Dieser hält alle Informationen des Interpreters. Sie erhalten einen gültigen State durch einen Aufruf von `luo_open()`.

Lua ist in C implementiert. Diese Beispielprogramme nutzen aus diesem Grund ebenfalls C.

```
Lua_state *state = lua_open(0);
if (state == NULL) {
    /* Error */
}
```

Wenn `lua_open()` NULL zurückliefert, dann ist irgendetwas wirklich schief gelaufen. Aber in der Regel können Sie davon ausgehen, dass der Aufruf glückt. Gehen Sie trotzdem immer auf Nummer Sicher und überprüfen Sie den Rückgabewert aller Funktionen.

Das kürzeste C-Programm, das Lua korrekt nutzt sieht in etwa so aus:

```
#include <stdio.h>
#include <lua.h>

int main(int argc, char** argv) {
    lua_State *state = lua_open();
    if (!state) {
        printf("Error opening Lua\n");
        return -1;
    }
    lua_close(state);
    return 0;
}
```

Dieses Programm macht nichts anderes, als einen Interpreter zu erzeugen und diesen dann wieder freizugeben.

Sie können dieses Programm nutzen um Ihre Installation von Lua zu testen. Wenn das Programm problemlos kompiliert, dann sind alle benötigten Bibliotheken und Header korrekt installiert worden

Lua-Standardbibliotheken

Der Großteil aller Funktionen ist in externen Bibliotheken von Lua implementiert, die Sie erst noch laden müssen. Zum Glück geht dies recht einfach. Jede Bibliothek hat Ihren eigenen Befehl zum Initialisieren.

```
lua_baselibopen(state);
lua_iolibopen(state);
```

```
    lua_strlibopen(state);
    lua_mathlibopen(state);
```

Nachdem die Bibliotheken geladen wurden, können Sie ein eigenes Skript durch den Aufruf von `lua_dostring()` ausführen.

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>

int main(int argc, char** argv) {
    lua_State *state = lua_open();
    if (!state) {
        printf("Error opening Lua\n");
        return -1;
    }

    lua_baselibopen(state);
    lua_iolibopen(state);
    lua_strlibopen(state);
    lua_mathlibopen(state);

    lua_dostring(state,
       "a = 1;\n"
       "b = 2;\n"
       "a = a + b\n"
       "print(a);\n"
    );

    lua_close(state);
    return 0;
}
```

Wenn Sie dieses Programm durch

```
gcc -o test1 test1.c -llua -llualib
```

kompilieren, und dann ausführen, sollten Sie in etwa folgende Ausgabe erhalten:

```
3
```

Und? Beeindruckt? Sie haben gerade eben zwei Variablen innerhalb eines Skriptes definiert, haben mit deren Werten gearbeitet und am Ende ein Ergebnis ausgegeben. Nicht schlecht, wenn man bedenkt, dass dafür nur 3 Aufrufe der Lua-Bibliothek nötig waren.

Script-Dateien

Der große Vorteil von Scriptsprachen ist ja bekanntlich, dass man eben nicht das Hauptprogramm neu erstellen muss, wenn man eine Änderung durchführt. Aus diesem Grund ändern wir nun das Beispielprogramm so ab, dass es ein Scriptfile von der Festplatte liest und ausführt.

```
#include <stdio.h>
#include <lua.h>
#include <lualib.h>

int main(int argc, char** argv) {
    lua_State *state = lua_open();
    if (!state) {
        printf("Error opening Lua\n");
        return -1;
    }

    lua_baselibopen(state);
    lua_iolibopen(state);
    lua_strlibopen(state);
    lua_mathlibopen(state);

    lua_dofile(state, "test.lua");

    lua_close(state);
    return 0;
}
```

Es ist nur der Aufruf einer einzigen Funktion nötig, um ein komplettes Scriptfile auszuführen. Also sehr viel einfacher geht es doch nun wirklich nicht mehr.

Hier ist das verwendete Test Skript, `test.lua`:

```
a=2;
b=a*3;
print (b+a);
```

Wenn Sie noch nie einen Interpreter von Hand geschrieben haben, werden Sie sich eventuell wundern, warum ein so simples Programm ein Glänzen in die Augen vieler Entwickler zaubert. Ein eigener Interpreter ist ein ziemlich großes Unterfangen, komplex genug um etliche Bücher damit zu füllen. Lua reduziert den Aufwand auf ein paar Zeilen Code.

Integration mit eigenem Code

Bisher hat der Lua Interpreter den Lua Code ausgeführt, und wir haben das Ergebnis über die Konsole ausgegeben.

Nun werden wir Lua-Funktionen von unserem Code aus aufrufen.

Luas Datentypen

Lua unterscheidet intern folgende Datentypen:

Name	Bedeutung
Nil	Ohne Inhalt
Number	Eine Zahl
String	Zeichenkette
Function	Funktionsaufruf
Userdata	Userspezifische Daten
Table	Tabelle von Werten

Tabelle 35.1: Luas Datentypen

Der Stack

Alle Daten von C zu Lua und umgekehrt werden über den Stack übergeben. Wenn Sie eine Funktion aufrufen, dann werden alle Parameter auf den Stack gelegt, Lua holt die Daten dann wieder vom Stack, führt die Funktion aus und legt das Ergebnis wieder auf den Stack.

Wichtig ist, dass positive Indices zum Stack eine absolute Position angeben, also zum Beispiel das 2. Element des Stacks. Diese Zahlen starten bei 1 und nicht bei 0, wie es in C üblich ist. Eine negative Zahl beschreibt die Elemente von der Oberseite des Stacks aus. Ein Wert von –1 ist also das oberste Element auf dem Stapel.

Doch grau ist alle Theorie. Nehmen wir an, Sie wollen die Allegro Messagebox (`allegro_message()`)-Funktion aufrufen. Dafür müssen Sie den zu übergebenden Parameter, sagen wir eine Zeichenkette, vom Stack holen und dann den Rückgabewert (-1 für Erfolg) wieder auf den Stack legen.

Diese Aufgabe erfüllt eine Schnittstellen Funktion:

```
int call_allegro_message(lua_State *state) {
    allegro_message(lua_tostring(state, -1));
    lua_pushnumber(state, 0);
    return 1;
}
```

Bevor diese Funktion von einem Lua Skript aufgerufen werden kann, müssen Sie dafür sorgen, dass sie dem Interpreter bekanntgegeben wird. Dies geschieht durch einen Aufruf von `lua_register()`.

```
lua_register(state,
             "allegro_message",
             call_allegro_message
);
```

Jetzt können Sie diese Funktion nach Belieben von Lua aus aufrufen.

Hier ist das Lua Testprogramm zum Aufruf der `allegro_message()`-Funktion:

```
a=2;
b=a*3;
allegro_message(b+a);
```

Der vollständige C-Code sieht wie folgt aus:

```
#include <allegro.h>

#include <stdio.h>
#include <lua.h>
#include <lualib.h>


int call_allegro_message(lua_State *state) {
    allegro_message(lua_tostring(state, -1));
    lua_pushnumber(state, 0);
    return 1;
}

int main(int argc, char** argv) {

    allegro_init();
```

```
    lua_State *state = lua_open();
    if (!state) {
        printf("Error opening Lua\n");
        return -1;
    }

    lua_baselibopen(state);
    lua_iolibopen(state);
    lua_strlibopen(state);
    lua_mathlibopen(state);

    lua_register(state, "allegro_message", call_allegro_message);
    lua_dofile(state, "test1.lua");

    lua_close(state);
    return 0;
}
END_OF_MAIN()
```

Aufrufen von Lua-Funktionen aus C

Ein Aufruf einer Lua-Funktion verläuft in 4 Schritten, wenn das Skript mit der Funktion bereits geladen wurde (zum Beispiel durch `lua_dofile()`).

- ✔ Die Lua Funktion muss auf den Lua Stack gelegt werden.
- ✔ Die Parameter müssen auf den Stack gelegt werden.
- ✔ Die Funktion wird aufgerufen.
- ✔ Die Rückgabewerte werden vom Stack genommen.

Angenommen wir haben in unserem Lua File eine simple Funktion wie diese:

```
function add(a, b)
    return (a+b)
end
```

Dann müssen wir erst die Funktion finden und auf den Stack legen, dann die zwei Parameter hinterherschicken, den Funktionsaufruf durchführen und schließlich den einzelnen Rückgabewert vom Stapel nehmen.

```
void test_lua_call(lua_State *state, int a, int b) {
    int result;

    lua_getglobal(state, "add");
```

```
    /* Aufruf einer Lua Funktion */
    lua_pushnumber(state,  a);
    lua_pushnumber(state,  b);

    /* 2 Parameter, 1 Rückgabewert */
    lua_call(state, 2, 1);

    result = (int) lua_tonumber(state, -1);

    /* Wert vom Stack nehmen */
    lua_pop(state, 1);
    allegro_message(
    "Called lua's add function with %i, %i\nresult:%i",
    a, b, result);
}
```

Das vollständige Programm sieht dann so aus:

```
 0);


ate *state, int a, int b) {

"add");

unktion */
  a);
  b);

kgabewert */
;

number(state, -1);

men */

ed lua's add function with %i, %i\nresult:%i", a, b, result);

 argv) {


a_open();

ning Lua\n");
```


Abbildung 35.1: Aufruf von Lua Funktionen aus C

```
#include <allegro.h>

#include <stdio.h>
#include <lua.h>
#include <lualib.h>
```

```
int call_allegro_message(lua_State *state) {
    allegro_message(lua_tostring(state, -1));
    lua_pushnumber(state, 0);
    return 1;
}

void test_lua_call(lua_State *state, int a, int b) {
    int result;

    lua_getglobal(state, "add");

    /* Aufruf einer Lua Funktion */
    lua_pushnumber(state,  a);
    lua_pushnumber(state,  b);

    /* 2 Parameter, 1 Rückgabewert */
    lua_call(state, 2, 1);

    result = (int) lua_tonumber(state, -1);

    /* Wert vom Stack nehmen */
    lua_pop(state, 1);
    allegro_message("Called lua's add function with %i,
%i\nresult:%i", a, b, result);
}

int main(int argc, char** argv) {

    allegro_init();

    lua_State *state = lua_open();
    if (!state) {
        printf("Error opening Lua\n");
        return -1;
    }

    lua_baselibopen(state);
    lua_iolibopen(state);
    lua_strlibopen(state);
    lua_mathlibopen(state);

    lua_register(state, "allegro_message", call_allegro_message);
    lua_dofile(state, "test2.lua");
```

```
    test_lua_call(state, 200, 20);


    lua_close(state);
    return 0;
}
END_OF_MAIN()
```

Ich hoffe, Sie haben Gefallen an Lua gefunden. Mehr Informationen über Lua, seine Syntax und genauere Informationen über den Einsatz in Ihren Programmen können Sie dem umfangreichen Handbuch und dem Lua Wiki entnehmen.

Das Wiki finden Sie auf *http://lua-users.org/wiki/*.

Der Quellcode zu diesen Programmen ist auf der Buch-CD zu finden.

36 Stimmen der Entwickler

In diesem Kapitel finden Sie Interviews mit bekannten Entwicklern von Allegro-Spielen, Allegro-Erweiterungen und sogar der eigentlichen Bibliothek.

Ich möchte allen hier interviewten Entwicklern meinen herzlichen Dank aussprechen, dass sie sich die Zeit genommen haben, meine Fragen so ausführlich zu beantworten.

Kenny »Sirocco« Thornton

Sirocco arbeitet seit 1997 an dem epischen Rollenspiel »Fenix Blade«, welches als »meist ersehntes Rollenspiel eines Hobby-Entwicklers« geführt wird. Er hat außerdem das Ballerspiel »Frenetic« geschrieben, in dem ein einzelner futuristischer Krieger mit seinem Kampfanzug gegen Horden von Feinden antritt.

Sie finden mehr Informationen (und natürliche die aktuellste Version dieser Spiele) auf seiner Webseite: *http://www.fenixblade.com/*.

Fangen wir doch ganz am Anfang an: Was war das erste Video- oder Computerspiel, das Sie gespielt haben?

Wenn ich mich recht erinnere, muss das »Combat« für den Atari 2600 gewesen sein.

Wie alt waren Sie, als Sie mit dem Programmieren anfingen?

Ich fing mit dem codieren an, als ich 6 Jahre alt war. Ich habe erst ein paar Jahre lang Textabenteuer Spiele geschrieben, bevor ich es dann von einem Tag zum anderen aufgab.

Das klingt, als ob Sie schon immer Spiele schreiben wollten. War das Ihre Motivation das Programmieren zu lernen?

Ich bin mir nicht sicher. Am Anfang habe ich sicherlich viel Zeit mit dem Codieren von Spielen verbracht, aber als ich dann später richtig zu programmieren anfing, kam ich nie auf die Idee ein Spiel zu schreiben.

Haben Sie je ein Spiel gespielt und dann das Gefühl gehabt, dass Sie »genau so ein Spiel, nur besser« machen sollten? Gab es Spiele, bei denen Sie dachten: »Wow! Das möchte ich auch können!«?

Nicht so richtig. Nachdem ich Dutzend oder mehr Rollenspiele auf verschiedenen Systemen (Amiga, PC, Super Nintendo) gespielt hatte, begann ich mit einem eigenen Entwurf, welcher die besten Eigenschaften aller RPGs die ich gespielt hatte verbinden sollte. Die größten Einflüsse kamen wohl von »Ultima IV« und den frühen »Final Fantasy« Spielen. Ich glaube mein Wunsch war es damals, die komplexe Welt der »Ultima« Reihe mit dem unglaublich leicht zugänglichen Spielprinzip der »Final Fantasy« Reihe zu verbinden.

Spielen Sie auch Rollenspiele mit Stift und Papier oder beschränken Sie sich auf Computer Rollenspiele?

Ich habe früher gerne mal »normale« Rollenspiele gezockt. Wir haben eine Zeitlang das gute alte »Advanced Dungeons and Dragons« (AD&D) gespielt, und dann ein paar weniger bekannte Rollenspiele wie »Paranoia« und »Star Frontiers«. Leider habe ich den letzten Jahren keine Gelegenheit mehr gehabt, an einem traditionellen Rollenspiel teilzunehmen.

Woher nehmen Sie Ihre Ideen? Gibt es eine bestimmte Sparte von Filmen und Büchern oder ein bestimmtes Genre, das Sie besonders inspiriert?

Ich kann mit Sicherheit sagen, dass einige literarische Werke einen Einfluss auf meine Spiele hatten – besonders was den Humor und die Charakterentwicklung angeht. Ich war immer der Meinung, dass es am interessantesten ist, einen Charakter zu spielen, dessen Beweggründe klar definiert sind und im Laufe des Spieles immer wieder betont werden.

Ich entwerfe hypothetische Situationen für meine Charaktere und überlege mir, wie sie in einer solchen Situation reagieren würden. Was würden sie tun, was würden sie auf alle Fälle vermeiden wollen? Einige dieser Situationen habe ich leicht verändert aus Büchern übernommen, andere wurden durch Filme angeregt. Nachdem ich dieses Gedankenexperiment ein paar Mal mit meinen Charakteren durchgespielt habe, kann ich Situationen und auch ganze Handlungsstränge einfach dadurch erzeugen, indem ich mir überlege, wie die einzelnen Figuren miteinander auskommen würden. Was passiert, wenn man ein paar von ihnen in einen Raum

setzt. Welchen Verlauf würde die Unterhaltung nehmen? Wie gut könnten die einzelnen Figuren einander leiden?

Durch diese Zusatzinformationen wird die Aufgabe die Handlung zu entwerfen eher einfacher, je mehr Informationen zusammen getragen werden.

Haben Sie eine bestimmte Methode, um Ihre Ideen zu sammeln und zu ordnen?

Wenn ich am Arbeiten bin, habe ich einen Stapel von Haftnotizen griffbereit, um Ideen, die mir spontan einfallen, zu notieren. Ideen, von denen ich denke, dass sie es wert sind aufbewahrt zu werden, kommen in einfache Textdateien, die ich nach der Art der Idee sortiere. Ideen, welche die Geschichte oder den Handlungsbogen betreffen, kommen in eine eigene Datei; technische Anmerkungen, Anpassungen, die ich noch zu machen habe, experimenteller Code und Checklisten gehen in jeweils eigene Dateien. Auf diese Weise kann ich meine Ideen einfach lesen und bearbeiten. Es ist ebenfalls kein Problem, sie von einem Rechner auf einen anderen zu übertragen. Und da sie im Textformat sind, spielt auch das Betriebssystems des Rechners, auf dem ich arbeite, keinen großen Unterschied.

Woran erkennen Sie, dass eine bestimme Idee sich gut für ein Spiel eignet?

Das ist eine kniffelige Frage. Beinahe jede Idee lässt sich in ein spaßiges und motivierendes Spiel verwandeln. Die Kunst dabei ist aber zu erkennen, bei welchen Ideen es Sinn macht, sie weiter zu verfolgen, und welche sofort eingestampft werden sollten. Wenn ich eine neue Idee habe, dann versuche ich zuerst herauszufinden, was die Aufgabe des Spielers sein wird, was er zu erreichen versucht. Dann überlege ich mir, welche Belohnungen er dafür erhalten könnte.

Oder, einfach gesagt, wenn ich eine Idee habe, in der es sich hauptsächlich um sehr viel Action dreht, dann erwarte ich nicht, dass der Spieler den ganzen Tag komplizierte Befehle eingibt. Die Kernfrage ist, was bei dem Spiel den Spaßfaktor ausmacht. Wenn Sie diese Frage nicht sicher beantworten können, oder wenn die Antwort einen schalen Beigeschmack hinterlässt, dann sollten Sie die Idee entweder weiter ausarbeiten oder vielleicht sogar einfach total vergessen.

Kommt es vor, dass einige der Ideen, die Sie im ersten Moment für großartig halten, am Ende einfach nicht zu funktionieren scheinen?

Sicher. Gelegentlich hatte ich ein paar im ersten Moment wundervoll anmutende Ideen – und dann merkte ich, dass es in dem Spiel aber auch rein gar nichts für den Spieler zu tun gab. Ich kann mich an keine Details erinnern, aber es kam vor.

Was beeindruckt Sie an einem Spiel am meisten?

Detail, Detail, Detail. Ich stehe auf Details. Ich liebe es in einer reichen, gut ausgearbeiteten Umgebung herumzuwandern. Besonders, wenn es ein hohes Maß an Interaktion mit dem Hintergrund gibt. Es ist harte Arbeit so viele Details hinzuzufügen, aber das Ergebnis ist es wert. Es ist aufregend sich durch ein Spiel zu bewegen und zu spüren, dass man wirklich die Idee des Designers begreift. Gerade bei Rollenspielen bevorzuge ich es, wenn Szenen vor Details gerade zu strotzen. Angefangen bei Gesprächen und anderen Interaktionen zwischen den Charakteren über das Hantieren mit Gegenständen im Hintergrund bis hin zu Dingen wie Fischen gehen zu können, oder sich selbst Gegenstände herstellen zu können. Spiele wie »Ultima VII« zeigen deutlich, dass es so etwas wie »zu viel Detail« nicht geben kann.

Viele Firmen veröffentlichen derzeit Remakes oder veröffentlichen ihre klassischen Spiele auf portablen Systemen. Glauben Sie, dass es zu wenig neue Ideen gibt?

Ich bin für Remakes, wenn das klassische Original wirklich eine Neuauflage verdient und das Entwicklerteam bereit ist der neuen Fassung auch den Grad an Aufmerksamkeit zu widmen, den sie verdient. Leider ist nicht jede Neuauflage so gut wie das Original. Der Wechsel von 2D zu 3D hat einigen Remakes in den letzten Jahren den Charme genommen.

Aber das Hauptproblem, das ich sehe, ist, dass Entwickler und Produzenten ihre erfolgreichen Titel ausschlachten. Obwohl Kreativität ein Opfer der aktuellen Lage in der Spielentwicklungsindustrie ist, und ich das von einem geschäftlichen Standpunkt auch verstehen kann, macht es das nicht leichter. Aber es gibt zum Glück Ausnahmen, die zeigen das es immer noch Kreativität im Spieldesign gibt.

Kommen wir zu Ihren Projekten. »Fenix Blade« ist das Rollenspiel von einem unabhängigen Entwickler schlechthin. Wie hat alles angefangen, was waren Ihre Ziele und was hat sich während der Entwicklung verändert?

Am Anfang war FB nur der Versuch ein Manuskript, an dem ich gearbeitet habe, in ein konsolenähnliches-Rollenspiel umzusetzen. Dabei musste ich dann natürlich die eigentliche Story gehörig verändern, um die überwältigende Melodramatik, die zu jedem Rollenspiel gehört, zu integrieren. Dies ist mir anfangs wirklich schwer gefallen.

Wie ich bereits anfangs sagte, die Idee war es ja, eine Rollenspiel-Chimera zu erschaffen, welche die besten Elemente aller Rollenspiele, die ich je gespielt hatte, mit meinen eigenen Ideen und Vorstellungen verbinden sollte. Das Hauptziel war es eine Welt zu erschaffen, die einen in Ihren Bann zieht, gefüllt mit Spielercharakteren die genauso gut ausgearbeitet waren wie die in meinem Manuskript. Am Ende des Spieles sollte der Spieler jeden Charakter so gut kennen, als hätten sie sein ganzen Leben mit ihm verbracht. Das war (und ist) ein ziemlich hoher Anspruch und eine Menge Arbeit – aber ohne Fleiß kein Preis.

Es könnte einige Leser geben, die Fenix Blade noch nicht kennen. Könnten Sie kurz etwas über die Handlung erzählen?

Wo soll ich anfangen? Nun, das Königreich Solancia ist in mitten seines zweiten Bürgerkriegs, und beide Seiten haben Schwierigkeiten in den Krieg zu ziehen wegen des massiven Gebirgszuges, welches das Land nahezu genau in zwei Hälften teilt. Es gibt eine einzige Route durch die Berge, den Engels-Pass, der auf beiden Seiten stark befestigt ist. Aus diesem Grund ist die Situation zu Beginn des Spieles ziemlich festgefahren.

Montag Coven, einer der spielbaren Charaktere in dem Spiel, gehört zu einer Splittergruppe namens Mirage. Diese existiert im Geheimen in der westlichen Hälfte des Landes, der Hälfte, die noch immer treu und loyal hinter der königlichen Familie steht. Da sie sich nicht mit der Ideologie ihrer Kameraden anfreunden können, bilden Sie ihre eigene Gruppe, um das Königreich zu retten, ohne auf unehrenhafte Taktiken zurückgreifen zu müssen.

Da sie ihre Operationsbasis auf einer großen Fregatte haben, steht es ihnen frei den Feind anzugreifen, ohne sich Sorgen über Vergeltungsmaßnahmen machen zu müssen, da Schiffe sehr selten in dieser Spielwelt sind. Leider kamen sie in Kampfkontakt mit einem Metallschiff ähnli-

cher Größe und waren gezwungen sich zurückzuziehen. Während sie sich auf der Flucht befinden, müssen Sie nun einen Weg finden ihren Plan durchzuführen und dabei ihren Angreifern immer einen Schritt voraus zu sein. Und hier beginnt die Demoversion, beinahe in der Mitte des Spieles.

Die Fenix Blade Demo strotzt geradezu von Details. Wie viel Zeit ist in die einzelnen Screens geflossen?

Das kommt wirklich darauf an, wie viel Detail der Raum braucht, um lebendig zu wirken. Einige Räume können in einer halben Stunde zusammengestellt werden, andere brauchen wiederum einen kompletten Tag, an dem neue Grafiken und Objekte erstellt werden müssen. Die Gegend um den »Sea of Tears« (See der Tränen) aus der zweiten Demo war nach 2 Tagen soweit, inklusive aller Grafiken. Das Script für den Endgegner und den Kampf wurden später hinzugefügt, was noch einmal ungefähr eine Stunde gekostet hat.

Die Stadt Berin hat einen ganzen Monat verschlungen, in erster Linie weil viel Zeit gebraucht wurde, um neue Objekte wie Aquarien, Gemälde und andere Einrichtungsgegenstände zu erstellen. Da ich aber diese Gegenstände nun auch in anderen Städten verwenden kann, gehen diese dann hoffentlich schneller.

Was wollten Sie an den bekannten Rollenspielen verbessern? Gibt es ein spezielles Feature, auf das Sie besonders stolz sind?

In wenigen Rollenspielen agieren die Charaktere wirklich mit ihrer Umgebung. Und dies finde ich nicht sehr befriedigend. Es ist ein nettes Detail, wenn Charaktere wirklich mit der Umgebung interagieren, also zum Beispiel in einen Spiegel sehen oder auch einfach mal nur kurz stehen bleiben, um einen umgestürzten Baum zu betrachten. Diese erlaubt es dem Spieler, einen besseren Einblick in die Figur zu erhalten, die er spielt. Wenn man sieht wie sich ein Charakter in einer bestimmten Umgebung verhält, ist es viel leichter, sich in die Gedankengänge und die Wünsche hineinzuversetzen. Und so kann der Spieler auch wirklich erleben, wie sich die Einstellungen der Spielfigur über die Zeit verändern. Charakterentwicklung ist ein wichtiger Punkt, und dabei meine ich nicht nur, ob er 20 oder 100 Punkte Schaden machen kann.

Die Hintergrundgeschichte von Fenix Blade kann wohl beinahe ein Buch füllen – wie schaffen Sie es, all diese Informationen zu verwalten?

Ich habe zahlreiche Notizen, sowohl auf Papier als auch in elektronischer Form. Leider kommt es aber auch zu Unachtsamkeiten. Ein Beispiel: Minx sollte ursprünglich Montag durch das »Northland Inn«-Szenario begleiten. Aber durch ihre extreme Abneigung zu kaltem Wetter wurde dies unmöglich. Obwohl ich mich wirklich bemühe solche Kleinigkeiten zu berücksichtigen, kann es passieren, das irgendetwas durch die Maschen schlüpft. Aber die Anmerkungen nach Themen zu sortieren hilft wirklich. Die meiste Zeit über kann ich mich jedoch auf mein Gedächtnis verlassen und die Notizen und Aufschriebe kommen nur dann zum Einsatz, wenn ich mich nicht mehr an ein technisches Detail erinnern kann (zum Beispiel eine Ereignisnummer oder eine Karte).

Neben der Story ist auch das Kampfsystem ein häufig diskutierter Teil eines Rollenspiels. Können Sie uns eine kurze Übersicht über die Besonderheiten des »Fenix Blade« Kampfsystems geben?

Die Methodik des Kampfsystems ist eine Hommage an das ausgeklügelte System von Final Fantasy 6. Ich habe mich dazu entschlossen, die allgemeine Ansicht beizubehalten und ebenfalls das Active-Time-Battle (ATB) System zu benutzen. Jedoch habe ich die Möglichkeiten der Akteure erweitert und den von den Menüs benötigten Platz so weit wie möglich minimiert.

Die Spieler behalten die Übersicht durch ein System von Aktionsindikatoren die über den Köpfen der Gegner angezeigt werden, kurz bevor diese angreifen. Dadurch kann der Spieler sich auf die Angriffe des Gegners einstellen, als ob er die Angriffe vorausahnen würde. Das ist so ähnlich wie bei einem Boxer, der aufgrund der Körperbewegungen seines Gegners dessen Aktionen abschätzen kann. Damit der Spieler gefordert wird und um die Kämpfe abwechslungsreicher zu machen, gibt es einige neue Elemente, die auch sehr gut mit den Aktionsindikatoren zusammenarbeiten.

Gegnerische Angriffe, die in einem kritischen Treffer enden können, geben dem Spieler für eine kurze Zeit die Möglichkeit einen Gegenangriff zu starten, indem sie eine bestimmte Abfolge von Kommandos eingeben, die einen mächtigen Konterangriff auslösen. Das Timing ist hier der Schlüssel, deswegen ist es immens wichtig den Fluss des Kampfes zu beachten.

Ein weiterer neuer Punkt sind die Ausweichbewegungen, die auf eine ähnliche Art und Weise funktionieren wie die Gegenangriffe. Der Spieler kann Angriffen ausweichen, wenn er den korrekten »Ausweichcode« ein-

gibt. Für jede Art des Angriffs gibt es einen eigenen Code. Der Spieler muss also auf die Anzeigen achten, damit er weiß, ob es zu einem körperlichen, magischen oder technischen Angriff kommt.

Jeder Charakter hat sowohl Fertigkeiten die er lernen und / oder andere lehren kann als auch ihm innewohnende Talente, die nicht übertragbar sind. Die meisten dieser Fertigkeiten und Talente können zusammen mit dem bekannten Magie System im Kampf eingesetzt werden.

Um dem Kampfsystem noch mehr Tiefe zu geben, hat jeder Charakter einen Ausdauerwert. Der Spieler wird ein Auge auf die verbleibende Ausdauer seiner Charaktere haben müssen, um die maximale Einatzbereitschaft der Gruppe sicherzustellen. Eine ganze Reihe von vernichtenden Angriffen kann den Ausdauerbalken sehr schnell erniedrigen und sie sehr verwundbar machen, bis sie sich wieder erholt haben. Dies gibt den erfahreneren Spielern die Möglichkeit einen schnellen Sieg herbeizuführen – mit dem Risiko möglicherweise ungeschützt dazustehen, falls der Angriff nicht erfolgreich sein sollte.

Spielbalance ist ein wichtiger Aspekt für ein Rollenspiel. Es ist ein häufiges Problem, dass ein zu schwacher Gegner versehentlich in Gebiete vordringt, in dem die Gegner zu stark für ihn sind, oder dass ein zu starker Spieler von schwachen Gegnern gelangweilt wird. Wie gehen Sie in »Fenix Blade« mit diesem Problem um?

Anstatt das Spiel dem Spieler anzupassen, gebe ich ihm Anreize sich in passendere Gebiete zu begeben. Die Erfahrungspunkte, die man für das Besiegen eines Gegners erhält, beruhen darauf, wie schwierig es für den Spieler ist einen bestimmten Gegner zu besiegen. Sobald die Charaktere also stärker werden, bekommen Sie weniger Erfahrung durch Kreaturen, die sie bereits kennen. Irgendwann muss man dann woanders hin reisen um überhaupt noch Erfahrungspunkte zu erhalten. Dadurch kann vermieden werden, dass ein mächtiger Charakter die Herausforderung aus dem Spiel nimmt.

Ist die Gruppe jedoch zu schwach, dann hat sie einige Möglichkeiten das Blatt zu ihrem Gunsten zu wenden. Talismane (»Charms«) können den Charakteren erstaunlich effektive Kampffertigkeiten verleihen und durch die gezielte Anwendung von Magie kann man bestimmte Monster sehr schnell ausschalten. Dazu kommt noch, dass geschickte Spieler die Aktionsindikatoren ausnützen können, um gegnerische Angriffe zu kontern oder ihnen auszuweichen. Eine schwache Gruppe mit durchschnitt-

licher Ausrüstung aber guten Fertigkeitswerten kann normalerweise gegen Angreifergruppen bestehen, die viel mächtiger sind als sie.

In den meisten RPGs sind Zaubersprüche viel mächtiger als Angriffe mit Waffen. Wie halten Sie die Balance zwischen Magie und Waffen?

Mit Zaubersprüchen ist es so eine Sache. Durch das ATB System kommt ein Timing Element in das Spiel. Dadurch kann es passieren, dass das Sprechen der mächtigsten Zaubersprüche so lange dauert, dass sie nicht mehr so nützlich gegen schwächere Gegner sind. Da die Charaktere mit der Zeit immer besser in der Beherrschung ihrer Waffen werden ist es möglich, dass sie recht schnell in der Lage sind, sehr viel Schaden in der Zeit zu verursachen, die es braucht den Zauberspruch vorzubereiten. Und dies ohne Magiepunkte zu verlieren. Da aber Zaubersprüche auch gegen mehrere Ziele eingesetzt werden können und stärkere Monster auch deutlich mehr Schaden vertragen, behalten die Sprüche ihre Nützlichkeit – ohne zur dominanten Lösung zu werden.

Auch verbrauchen Zaubersprüche Mana. Da alles, was Mana auffüllen kann, im Spiel sehr teuer ist, müssen die Charaktere Gegenstände suchen, die ihnen helfen, ihre Zauberpunkte aufzufüllen. Einfach in einen Laden zu gehen wird sehr schnell sehr teuer.

Könnten Sie uns sagen, wie viele verschiedene Monster, Nicht-Spieler-Charaktere (NPCs), Waffen, Rüstungen und Zaubersprüche Fenix Blade hat?

Derzeit gibt es 98 Monster, 118 NPCs, 489 Waffen / Gegenstände / Talismane / Rüstungen im Spiel und etwa 100 Zaubersprüche. Diese Zahlen können sich natürlich noch in der weiteren Entwicklung verändern. So ist es sehr gut möglich, dass noch weitere NPCs dazukommen.

Wie viel Zeit ist bisher in »Fenix Blade« geflossen? Haben Sie damit gerechnet, dass die Entwicklung des Spiel so zeitaufwendig wird?

Ich kann absolut nicht sagen, wie viel Zeit bereits in »Fenix Blade« geflossen ist. Sagen wir einfach mal, dass ich seit 1997 ein sehr beschäftigter Bursche bin. Ich habe immer gewusst, dass ich mir da einiges vorgenommen habe und war auch auf eine lange Entwicklungszeit vorbereitet – aber um ehrlich zu sein, habe ich schon angenommen, dass ich weniger als 7 Jahre brauchen würde.

Ich nehme an, Sie hassen diese Frage, aber wann wird »Fenix Blade« abgeschlossen sein? Und: Wird es eine weitere Demo geben?

Das kann nur die Zeit beantworten. Wenn wir davon ausgehen, dass ich genug Zeit habe daran zu arbeiten, und meine Nebenprojekte abgeschlossen sind, dann denke ich schon, dass ich es schaffe in den nächsten zwei Jahren damit fertig zu werden. Aber wir alle wissen, wie unberechenbar das echte Leben sein kann. Demo III wird einen oder zwei Monate vor der endgültigen Version des Spiel zum Download bereit stehen.

»Frenetic« ist so in etwa das genaue Gegenteil von »Fenix Blade«. Brauchten Sie eine Auszeit von Ihrem normalen Entwicklungstrott?

»Frenetic« war eine gute Weise drei Wochen Urlaub am Stück zu füllen und einige Experimente im Bereich Game Design innerhalb eines sehr engen Zeitrahmens durchzuführen. Ich habe eine Geschwindigkeitsänderung gebraucht, und als sich die Gelegenheit bot, habe ich zugegriffen. Ich bin ziemlich froh, dass sich die Dinge so entwickelt haben wie sie es taten. Die ganze Zeit an »Fenix Blade« zu arbeiten gibt einem wenig Spielraum für Experimente, und das ist etwas, was ich wirklich vermisse. »Frenetic« und »Frenetic Plus« haben es mir erlaubt, einige interessante Nebenstrecken zu erforschen.

Kommen wir auf Projekte zu sprechen, über die bisher nur wenig bekannt ist. Was ist »Cry Havoc«?

Bei »Fenix Blade: Cry Havoc« handelt es sich um einen Versuch die Welt von »Fenix Blade« mit den taktischen Rundenkämpfen und der knappen, ernsthaften Kampagnenpräsentation von Kriegsspielen wie »Ghost Recon« zu verbinden. Der Schwerpunkt wird ganz klar bei den taktischen Elementen liegen und nicht so sehr auf dem Erreichen neuer Erfahrungsstufen und dem Betrachten von Zwischensequenzen. Ähnlich wie beim Schachspiel wird es nur wenige Regeln geben, aber viele Dinge, die man in Betracht ziehen muss. Auch wird das Gelände eine wichtige Rolle in den Kämpfen einnehmen.

Wann werden wir eine Demo Version von »Cry Havoc« spielen können?

Es ist geplant, dass »Fenix Blade: Cry Havoc« im Januar 2004 abgeschlossen sein wird, wenn nicht sogar früher. Da mehr Wert auf das Spielprinzip gelegt wird, wird weniger Zeit für das Erstellen von Animationen und anderem Schnick-Schnack benötigt.

Gibt es Projekte, die Sie bisher noch nicht angekündigt haben? Können Sie uns irgendwelche kleinen Geheimnisse verraten?

Ich arbeite gerade an den ersten Entwürfen für ein Minispiel namens »Stanza«, in dem es darum geht, dass Montag gegen mythologische Kreaturen im Musikwettstreit antritt. Ein seltsames Konzept, aber es scheint ganz gut zu funktionieren. Ich hoffe, ich finde dieses Jahr noch etwas Zeit daran zu arbeiten.

Welchen Rat würden sie einem Neuling in Sachen Rollenspielentwicklung mit auf den Weg geben?

Hört auf die Leute, die bereits schon länger an Rollenspielen arbeiten als Ihr. Werft einen Blick auf die Leute, die nach euch angefangen haben, ein solches Spiel zu entwickeln. Die Chancen stehen gut, dass Ihr von beiden Gruppen etwas lernen könnt. Und haltet euch vor Augen, dass Ihr die Regeln aufstellt. Bringt neue Ideen ein, wo es möglich ist, und benutzt vorhandene, wo es nötig ist. Und schließlich: Es gibt nur eine Sache, auf die es bei einem Spiel wirklich ankommt, und das ist, dass es Spaß macht.

Paul Pridham

Paul arbeitet derzeit an »Super Action Kung Fu« (SAKFU), einem sehr Arcade-mäßigen 2D Prügelspiel mit Comic Figuren als Akteuren. Nach der ersten Demoversion wurde SAKFU zu einem der am sehnlichst erwarteten Allegro Spiele.

Hallo Paul! Können Sie sich noch an Ihr erstes Computer- oder Videospiel erinnern?

Oh. Das ist eine schwere Frage – ich kann mich nicht mehr genau erinnern. Aber ich denke, das erste Videospiel war eines dieser mechanischen Ballerspiele, in dem das Bild eines Haifischs oder eines anderen Gegners auf einen Schirm projiziert wurde. Das Ziel war es, den Gegner mit der unförmigen »Waffe«, die am Gehäuse befestigt war, zu treffen.

Das erste Spiel, dass ich an einem Bildschirm gespielt habe, war sicherlich ein Vektorspiel wie »Asteroids«.

Ich erinnere mich auch noch daran bei meiner Cousine »Pong« gespielt zu haben – aber ich war damals so jung, dass ich mich nicht mehr erinnere, welches Spiel ich jetzt wirklich zuerst gezockt habe.

Sagen wir einfach: Ich habe Videospiele gespielt, seit es sie gibt (grinst).

Warum haben Sie mit dem Programmieren angefangen? Wollten Sie schon immer Spiele entwickeln, oder hatten Sie zu Anfang einen anderen Beweggrund?

Als ich etwa 10 Jahre alt war, fing ich an auf einem Commodore 64 zu programmieren, den mein Bruder und ich zu Weihnachten bekommen hatten. Ich habe ganz bestimmt zu codieren angefangen, weil ich Spiele erschaffen wollte – das Programmieren war nur ein Mittel zum Zweck.

Gibt es ein paar Spiele, die Sie zu eigenen Spielen inspiriert haben?

Ich denke, dass viele der Spiele, die ich spiele, mich in der einen oder anderen Weise motivieren. Entweder weil die Entwickler eine gute Idee hervorragend umgesetzt haben, oder weil sie es geschafft haben, eine tolle Idee komplett in den Sand zu setzen. Manchmal kann man aus Fehlern mehr lernen als aus Dingen, die richtig gelaufen sind. Fehler springen einem deutlicher ins Auge.

Haben Sie eine besondere Methode Ihre Ideen zu sammeln? Gibt es einen besonderen Kniff, den Sie weitergeben möchten?

Die Ideen überkommen mich meist ziemlich ungeplant in den seltsamsten Momenten. Wenn mir eine Idee dann wirklich zusagt, dann schreibe ich Sie mir in mein Notizbuch. Ich habe auch einen Wiki Server auf meinem lokalen Rechner laufen, den ich nutze, um meine Ideen zu ordnen.

Das Gute am Niederschreiben von Ideen ist, dass man sich, sobald sie auf dem Papier festgehalten sind, nicht weiter um sie zu kümmern braucht, sondern sich wieder auf das aktuelle Problem konzentrieren kann.

Viele Leute machen den Fehler ihren Ideen sofort nachzujagen und dabei dann die aktuellen Projekte zu vergessen.

Woher wissen Sie, ob sich eine Idee zu einem Spiel umfunktionieren lässt?

Das ist einfach: Ich spiele das Spiel im Geiste. Wenn ich dabei Spaß habe und nach einer Runde Gedankenspiel mit einem Grinsen vor meinem Rechner sitze, dann weiß ich, dass die Idee gut ist.

Ist es schon einmal vorgekommen, dass eine Idee anfangs sehr gut wirkte, sich dann aber einfach nicht gut umsetzen lies?

Sicherlich. Aber ich denke, das kommt daher, dass die Idee einfach noch nicht ausgearbeitet genug war. Ich glaube, dass man alles in ein witziges Spiel verwandeln kann, wenn man sich nur die Zeit nimmt, lange und ausführlich an der Idee zu feilen. Man muss den Kleinigkeiten Beachtung schenken. Meist sind es die kleinen Dinge, die einen großen Unterschied ausmachen.

Welche Art von Spielen gefällt Ihnen am besten?

Ich bin ein großer Fan von Abenteuerspielen. Allerdings meine ich damit nicht die üblichen »Point-and-Click«- und Erkundungsspiele. Es ist mehr das Abenteuer, das durch das Gameplay selbst erzeugt wird, wenn man Dinge ausprobiert, Sachen entdeckt und verschiedene Möglichkeiten ausprobiert. Ich bin kein großer Freund der linearen Ein-Raum-nach-dem-anderen-Abenteuer.

Viele der aktuellen Spiele waren in einer ähnlichen Form schon einmal da. Gehen den Spieldesignern die Ideen aus?

Es gibt so viele Ideen, die man ausprobieren kann, wie das Universum groß ist. Allerdings muss man mit alten Ideen und Konventionen brechen und Althergebrachtes von Innen nach Außen stülpen, wenn man wirklich innovativ sein möchte.

Ich glaube, die Leute erinnern sich an die alten Spiele, weil die Ideen damals frischer waren. Damals ging es noch nicht so sehr um das Marketing, die Entwickler beschränkten sich nicht nur auf die Genres, die sich am besten verkauften.

Natürlich ist ein weiterer Grund, dass die Jugendlichen von damals heute kaufkräftige Kunden sind und gerne etwas Geld für einen Hauch Ihrer Jugend opfern.

Kommen wir zu SAKFU. Dieses Spiel macht den Eindruck eines klassischen Kung-Fu-Films, der allerdings von einem Cartoon Studio umgesetzt worden ist. Haben die Mythen der übermächtigen Shaolin Mönche Sie inspiriert? Gibt es irgendwelche Mythen, Legenden oder Filme, die einen Einfluss auf SAKFU hatten?

In der Tat wurde »Super Action Kung Fu« von den Kung-Fu-Filmen der 70er Jahre inspiriert. Auch Spiele wie »Yie-Ar Kung Fu« und »Shao-Lin's Road« haben sicherlich einen Eindruck hinterlassen.

SAKFU ist eigentlich ein Remake eines alten Spieles, das ich früher auf meinem C64 angefangen habe, aber aufgrund meiner damals noch nicht so ausgereiften Programmierkenntnisse nicht fertigstellen konnte.

Ich liebe den Stil und das Flair der alten Kung-Fu-Filme und habe versucht, möglichst viel davon in diesem Spiel einzufangen.

Wie haben Sie all die verschiedenen Kämpfer entworfen?

Normalerweise lasse ich einfach meiner Vorstellungskraft freien Lauf und denke mir Begebenheiten und Umgebungen aus. Natürlich gibt es auch äußere Einflüsse, ein paar Charaktere sind Helden und Schurken aus Kung-Fu-Filmen, die ich besonders gut fand, nachempfunden.

Obwohl die Sprites in SAKFU über Häuser springen können, fühlt sich das Spiel immer noch recht realistisch an. Die eigentlichen Kampfaktionen könnten auch von einem sterblichen Karateka oder Kung-Fu-Kämpfer ausgeführt werden. Wie haben Sie die Angriffe der einzelnen Sprites festgelegt? Haben Sie selbst Kampfsporterfahrung?

Nun, nachdem die meisten Charaktere nur recht wenige Moves, haben war das nicht sehr schwer (grinst). Ich habe einfach versucht, jeden Charakter so einzigartig wie möglich zu gestalten. Dies gilt sowohl für das Aussehen, als auch die Art und Weise, wie sie sich im Spiel anfühlen.

Wenn es nur wenige Moves gibt, dann ist dies natürlich deutlich einfacher, da dadurch jede einzelne Bewegung viel wichtiger wird. Wenn es für jeden Charakter 30 Bewegungen gäbe, die jede Richtung abdecken würden, dann würde dies die einzelnen Figuren zu ähnlich machen.

Oh, und ja, ich habe etwas Erfahrung in den Kampfkünsten, würde mich aber keinesfalls als Experten bezeichnen.

Obwohl SAKFU ein Prügelspiel ist, erscheint es nicht gewalttätig. Eigentlich habe ich sogar den Eindruck, dass in den Zeichentrickserien die Samstag morgens laufen, mehr Gewalt vorkommt.

Ich wollte einfach nur ein Spiel schreiben, das Spaß macht. Blutspritzer waren deswegen einfach nicht so wichtig für mich.

Ich denke schon, dass der Einsatz von Blut in manchen Spielen durchaus den Spielspaß oder die Spannung steigern kann – ich bin allerdings auch der Meinung, dass SAKFU nicht zu diesen Spielen gehört.

Es ist mir auch schwer gefallen, meine süßen kleinen Sprites zu verstümmeln. In einer perfekten Welt könnten wir alle durch die Luft springen und Kung-Fu ausüben, ohne dabei verletzt zu werden.

Würde man SAKFU in ein Arcade-Kabinett stecken, würde niemand bemerken, dass es sich um ein PC-Spiel handelt. Haben Sie SAKFU von Anfang an so geplant?

Es hat sich einfach so ergeben. Ich hatte die Idee von diesen kleinen süßen Leuten, die durch die Gegend hüpfen und dabei Kampfschreie von sich geben. Es gibt hier keine geheimen Spiel-Design-Mysterien die ich aufdecken könnte. Sorry (grinst).

Was heißt hier keine Mysterien? Wie schaffen Sie es, an einem normalen PC, ohne besondere Eingabehardware ein perfektes Spielhallengefühl zu erschaffen?

Ich versuche einfach, alles möglichst simpel zu halten. Es hilft natürlich auch, wenn man einige Bewegungen aus dem derzeitigen Zustand der Spielfigur ableitet. Dann hat man einen neuen Move, ohne dafür eine weitere Taste opfern zu müssen. Und das Spiel fühlt sich dadurch auch viel lebendiger und weniger statisch an.

Wie würden Sie »Arcade Gameplay« beschreiben?

Nun das ist einfach! Arcade Gameplay ist einfach zu lernen, man braucht nicht lange, um die Kontrollen zu begreifen, und das Spiel mach Spaß solange es andauert. Man braucht nicht mehrere Stunden, um zu begreifen wie man ein Spiel in der Arcade benutzen muss, und es dauert nicht 100 Stunden oder mehr bis man es durch hat.

Wenn Sie die klassischen Prügelspiele wie »Karateka«, »Shao-Lin's Road« und »Way of the Exploding Fist« mit denen der neueren Generation vergleichen – wo sehen Sie die Hauptunterschiede?

Ich denke der Hauptpunkt ist wieder die Kontrolle der Spielfigur. Bei den neuen Spielen muss man Joystick Kung-Fu beherrschen, um die wirklich mächtigen Moves ausführen zu können. Dies ist sicherlich eine Sache, die viele Fans dieses Genres anspricht – allerdings verkaufen sich die leichter zugänglichen Spiele besser. Was ja auch Sinn macht. Denn bei leichter zugänglichen Spielen kann man auch gewinnen ohne, ein

Hardcore Gamer zu sein – und die meisten Leute da draußen sind keine Hardcore Gamer.

In den aktuellen Spielen wird auch häufig die Umgebung in den Kampf mit eingezogen. In den Prügelspielen, in denen man Monster kontrolliert, kann man mit Häusern und Autos nach dem Gegner werfen. In den normalen Spielen ist es möglich, dass ein Kämpfer durch eine Wand in völlig neues Gebiet bricht. Wie stehen Sie dazu?

Ich finde das Klasse. Dies ist in der Tat eines der besten Elemente der aktuellen Generation an Prüglern. Und es trifft genau meine Idee von Abenteuer. Ich kann mit der Welt interagieren, Dinge ausprobieren.

Dem Spieler zu erlauben Dinge zu tun, die nicht entscheidend für das eigentliche Spiel sind, können das Gameplay deutlich verbessern.

Planen Sie bereits SAKFU2? Wenn ja, was können Sie uns darüber erzählen?

Aber sicher. SAKFU2 wird all die Elemente und Ideen enthalten, die mir bei der Entwicklung von SAKFU gekommen sind. Es wird dem normalen SKAFU sehr ähnlich sein – aber mit noch mehr Spieltiefe, mehr Dingen, die man entdecken kann, noch mehr Sachen, die man zerstören kann.

Es wird das Spiel werden, dass ich als Kind immer spielen wollte, aber damals keiner programmiert hat.

Noch eine abschließende Frage: Was sind Ihre Empfehlungen für die jungen Designer und Programmierer?

Bleibt euch treu. Zweifelt nicht daran, dass ihr es schaffen könnt. Wenn Ihr eine gute Vorstellung von dem habt, was ihr erreichen wollt, dann ist alles andere einfach nur ein Mittel zum Zweck. Compiler, Zeichenprogramme, Editoren, Script Sprachen und Entwicklerforen.

Wenn Eure erste Idee gewaltig ist, dann schreibt sie auf und arbeitet an etwas, das kleiner und einfacher ist – aber der eigentlichen Idee immer noch nahe kommt. Das bringt Euch näher an Euer Ziel – und Ihr könnt das Projekt beenden bevor Ihr frustriert aufgebt, weil Ihr mehr abgebissen habt als Ihr kauen könnt. Sich mit den ersten Projekten zu übernehmen ist ein Fehler, der zu häufig gemacht wird.

Nutzt die kleinen Projekte, um das zu lernen, was Ihr für Euren Traum benötigt.

Ben Davis

Ben Davis ist die treibende Kraft hinter DUMB und der Autor von Spielen wie »Sheep«, »Balls«, »Ibiza Insurgency« und »Set Up Us The Bomb !!!«.

Er ist ein regelmäßiger Poster auf *http://www.allegro.cc/* und eine der Informationsquellen auf dem Allegro IRC Channel.

Hi Ben!

Hallo! Bevor wir anfangen wollte ich erwähnen, dass »Set Up Us The Bomb!!!« ein Team Projekt war – ich möchte mich hier nicht mit fremden Federn schmücken (lacht).

Ich würde bei Dir gerne mit der gleichen Frage anfangen, die ich den anderen auch schon gestellt habe: Was war Dein erstes Computerspiel?

Wahrscheinlich war das »Repton 3« auf dem BBC Micro. Ich habe es beinahe unablässig gespielt – allerdings nur selten die Level, die bei dem Spiel dabei waren. Das große Plus bei diesem Spiel war der mitgelieferte Karteneditor, mit dem man eigene Level und sogar eigene Grafiken entwerfen konnte. »Repton 3« war übrigens auch ein großer Einfluss für »Rock'n'Spin«.

In welchem Alter haben Sie mit dem Programmieren begonnen?

Wenn ich ehrlich sein soll: Keine Ahnung. Ich glaube ich war 7 oder 8, aber ich kann mich wirklich nicht mehr erinnern. Allerdings weiß ich noch, wie mir mein 2 Jahre älterer Bruder ein BASIC Programm mit 5 Zeilen zeigte und mir erklärte, wie es funktioniert.

Die ersten Programme, die ich geschrieben habe, waren alles Menü Systeme. Ich habe einige davon geschrieben. Das war recht einfach, da sie alle nur ein paar PRINT Befehle und eine Tastaturabfrage benötigten.

Ich habe auch ein paar Programme geschrieben, die ein Lied spielten, indem ich haufenweise `SOUND` Anweisungen in das Programm gehackt habe. Kein sehr schöner Programmierstil, aber es hat funktioniert und mir Spaß gemacht.

Ich habe auch ein paar kleine Zeichenprogramme geschrieben, inspiriert durch ein Tool namens »Storyboard Plus«.

Für ein Kind sind Farben, Grafiken, Töne und Spiele einfach interessanter als Textwüsten. Ich denke, das ist der Grund, warum ich auch noch kein Textverarbeitungsprogramm geschrieben habe.

Aus dem gleichen Grund schreibe ich heute Spiele. Es ist nicht nur das Codieren, auch das Erstellen von Grafiken und Musik reizt mich sehr.

Gibt es ein Spiel, das Sie zur Nachahmung angeregt hat?

Vor langer Zeit wollte ich einmal das Spiel »Pipe Mania«, das ich auf dem BBC Micro gesehen habe, nachbauen. Allerdings hatte ich damals noch keine Ahnung wie und gab schnell wieder auf.

Aber es gab auch andere Spiele, die mich inspiriert haben, »Bomberman« und »Worms« zum Beispiel.

Wie bekommen Sie die Ideen für Ihre Spiele? Gibt es bestimmte Medien, die Sie beim Sammeln von Ideen bevorzugen?

Am meisten beeinflussen mich sicher andere Spiele. Meine besten Spiele sind Remakes von alten Spielen, deren Ideen mich überzeugt haben. Ich muss am Spieldesign noch arbeiten. Die Spiele, die ich komplett alleine entworfen habe machen zwar für einige Zeit Spaß, aber ich finde die Langzeitmotivation könnte besser sein.

Gab es eine Spielidee, die genial klang, aber im Spiel selbst dann nicht richtig wirkte?

Die Bewegung der Pods in PodFight gehört eventuell in diese Kategorie. Der Pod bewegt sich ununterbrochen, wenn man ihn nicht stoppt. Er hört nicht einmal von selber auf sich zu drehen. Ich fand diese Idee klasse, aber es macht das Spiel sehr schwer zu spielen.

Module-Files sind MIDI-Files sehr ähnlich, allerdings speichern Sie die Sample Daten des Instrumentes gleich mit ab. Ein Module sollte also auf jedem Rechner gleich klingen, verbraucht aber mehr Platz als MIDI-Dateien.

Was hat Sie dazu bewogen eine Bibliothek zu schreiben, welche die alten Amiga Musik Files wiedergeben kann?

Ich habe einen Musik Editor in Quick Basic und Assembler. Verglichen mit einem Programm wie, sagen wir, Impuls Tracker war es natürlich nicht sehr atemberaubend, aber nichts desto trotz konnte ich damit einige Musikstücke schreiben, die sich gar nicht mal so schlecht angehört haben.

Im Sommer 2001 habe ich mich dazu entschlossen einen weiteren Editor zu schreiben, aber diesmal einen, der deutlich mehr kann. Das war die Geburtsstunde von DUMB, da ich ja nicht nur einen Editor brauchte, sondern auch eine Bibliothek, um die Stücke abzuspielen.

Ich entschloss mich die Musikstücke in ».duh« Dateien zu speichern und entwickelte ein Dateiformat für DUH Files. Um die Routinen zu testen entschloss ich mich einen Konverter für die .IT Files von Impulse Trakker zu schreiben. Das funktionierte sehr gut, allerdings sind die Formate sehr unterschiedlich und die vom Konverter erzeugten DUH Files waren deutlich größer als die IT Dateien. Das Ende vom Lied war, dass ich den Konverter aufgab und mich entschloss die IT Files einfach zu laden.

In anderen Worten: Der Teil von DUMB, der die Module Dateien abspielt, ist eigentlich rein zufällig entstanden. Aber ich habe weiter an ihnen gearbeitet und sie wieder und wieder verbessert. Zum einen hat das Spaß gemacht, zum anderen war es eine sehr nützliche Funktionalität.

Wenn Ihr euch jemals gefragt habt, warum die API von DUMB so seltsame Begriffe wie »signrenderer« benutzt, dann liegt das daran, dass sie dafür ausgelegt ist mehr als die normalen Module abzuspielen. Die API passt hervorragend zu der (geplanten) Struktur der DUH Dateien.

Wird der Editor auch Plattform-übergreifend sein?

Absolut. Nach langen Debatten darüber, welche GUI Bibliothek nun verwendet werden soll, habe ich mich für QT von Trolltech entschieden. Wenn ich bei dieser Entscheidung bleibe, dann wird der Editor für alle Plattformen erhältlich sein, die von QT unterstützt werden. Dazu gehören unter anderem Linux und Windows.

Aber da ich an diesem Projekt in meiner Freizeit arbeite kann ich keine Versprechungen bezüglich eines Erscheinungsdatums machen.

Bei Kompressionsformaten wie OGG und MP3 – brauchen wir da eigentlich noch Module Files?

Module sind immer noch deutlich kleiner als MP3 und OGG Dateien – vor allem wenn man Wert auf gute Klangqualität legt. Und die Dateigröße ist nicht der einzige Vorteil gegenüber Ogg Vorbis (.ogg) und MPEG-1 Layer 3 (.mp3) Dateien. Da Module in Echtzeit gerendert werden, kann man als Programmierer einiges mit ihnen anstellen. Man kann einzelne Kanäle stumm schalten, Instrumente austauschen, die Geschwindigkeit verändern und so weiter.

Auch darf man nicht vergessen, dass MPEG-1 Layer 3 und Ogg Vorbis verlustbehaftet sind. Selbst wenn Sie den Unterschied nicht hören können – die Chancen stehen gut, dass es jemanden gibt, der es kann. Module Files sind eine Art »Quellcode für Musik«. Sie sind verlustfrei (solange der Player keine Fehler beim Abspielen macht). Das könnte auch bedeuten, dass man Module nutzt, um die Musik zu erzeugen – auch wenn das Ausgabeformat MP3 oder OGG ist.

Auch sorgt die Tatsache, dass es eine Menge guter Musik im Module Format gibt, dafür, dass gute Abspielsoftware für diese Stücke noch lange gefragt sein werden.

Oh – noch etwas in eigener Sache: Wenn Sie Ogg Vorbis nicht kennen oder gerade überlegen ob Sie OGG anstelle von MPEG-1 Layer 3 nehmen sollen – ich würde Ogg Vorbis empfehlen. Bei gleicher Kompression klingen die meisten Stücke als Ogg Dateien deutlich besser als MP3 codiert. Ein weiterer Vorteil ist, dass Ogg frei von Patenten ist.

Wie kam es zum Namen der Bibliothek? Der Name DUMB ist doch recht ungewöhnlich...

Ich bin mir nicht mehr 100% sicher, aber ich glaube es war eine Entscheidung aus einer Laune heraus. Ursprünglich war der ausgeschriebene Name »Dedicated Universal Music Bastardization.«

Der Name hat sich zwischenzeitlich in »Dynamic Universal Music Bibliotheque« geändert. Das macht die ganze Sache politsich korrekter.

Gibt es Pläne OGG komprimierte Module Dateien in zukünftigen DUMB Versionen zu unterstützen?

Wahrscheinlich schon. Wenn ja, dann wird dieses Format wahrscheinlich im DUH Format gespeichert. Aber ich möchte erst mal sicherstellen, dass die aktuellen Formate fehlerfrei unterstützt werden.

DUMB wird bereits im gleichem Atemzug mit FMod, einer im professionellen Betrieb genutzten Bibliothek für Module Dateien, genannt. Wenn Sie DUMB mit den anderen Playern und Bibliotheken vergleichen, wo liegen Vorteile von DUMB?

Es gibt zwei große Vorteile, die DUMB gegenüber FMod besitzt. Der erste Vorteil ist Genauigkeit. FMod spielt zwar XM Dateien korrekt ab, macht aber Fehler beim Abspielen von IT Files. Es unterstützt auch einige der Impulse Tracker Filter nicht – und die gehören zu meinen Lieblingsfeatures. Der einzige Player, den ich kenne, der Module Dateien ebenso korrekt wie DUMB wiedergibt, ist BASS. Aber BASS ist leider keine Open Source Software.

Der zweite Vorteil von DUMB ist, dass seine Lizenz dem Entwickler viel mehr Freiheiten gibt. Selbst kommerzielle Entwickler können DUMB kostenlos benutzen, solange sie nicht behaupten, Sie hätten den Player selbst entwickelt.

Der Quellcode ist frei verfügbar, jeder kann also Änderungen vornehmen (wenn das denn nötig sein sollte) oder überprüfen, auf welche Art und Weise manche Features funktionieren.

Der Source Code von FMod ist nicht frei verfügbar, und Sie dürfen FMod nur dann kostenlos benutzen, wenn Ihr fertiges Produkt auch kostenlos vertrieben wird.

FMod bietet jedoch auch einige Vorteile. Die Mixerfunktionen sind deutlich schneller. Dies ist allerdings nicht überraschend, da DUMB derzeit noch in purem C geschrieben ist und keine optimierten Assembler Routinen benutzt.

Auch ist FMod ein vollständiges Sound System, DUMB setzt hingegen auf einem bestehendem System (wie dem von Allegro) auf.

Gibt es einen modernen Tracker, der DUMB benutzt?

Soweit ich weiß nicht. Ich hatte ein paar Nachfragen von Leuten, die etwas in dieser Richtung programmieren wollten, aber bisher habe ich von keiner tatsächlichen Umsetzung gehört.

DUMB Studio ist allerdings schon auf dem Weg. Allerdings wird es nicht dazu ausgelegt sein IT Dateien oder andere bestehende Formate zu erzeugen. Das DUH Format wird recht unterschiedlich sein. Es wird Import Filter geben, die es einem erlauben ein Modul zu laden, zu ändern und das Ergebnis als DUH zu speichern.

Kommen wir von DUMB zurück zu Spielen. Die Liste Ihrer Spiele ist recht beeindruckend. Welches Spiel halten Sie persönlich für Ihr bisheriges Meisterwerk, und warum?

Das ist recht schwer – aber ich glaube mir gefällt Rock'n'Spin am besten. Leider ist es ziemlich alt, besteht aus furchtbaren Code und kann nur schwer von DOS auf andere Plattformen portiert werden. Aber ich glaube dieses Spiel hat am meisten Inhalt.

Wenn Sie an diesem Spiel nachträglich etwas ändern könnten, was wäre das?

Die Tastatur-Steuerung. Ich denke, sie kann recht verwirrend sein.

Arbeiten Sie derzeit an einem Spiel? Wenn ja, was können Sie uns darüber erzählen?

Mein derzeitiges Spieleprojekt ist streng geheim. In der Tat habe ich schon häufiger von meinem »SEKRIT PROJACT« gesprochen – und wenn ich es veröffentliche, werde ich diesen Codenamen auch irgendwo erwähnen, damit allen klar ist, dass es dieses Spiel war, das ich so lange geheim gehalten habe.

Wenn Sie es gar nicht mehr aushalten, dann laden Sie sich eine Version von Pod Fight herunter. In der `readme.txt` gibt es einige Informationen über mein geheimes Projekt.

Sie nehmen regelmäßig an Wettbewerben wie dem »Speedhack« teil. Was macht diese Ereignisse so besonders?

Es fällt mir manchmal schwer mich selbst zu motivieren. Ein Spiel zu schreiben dauert Monate. Speedhack macht Spaß, da man eine feste Deadline hat. Nach 76 Stunden ist es vorbei. Ich kann mich in die Aufgabe verbeißen und werde durch den Zeitdruck gezwungen auch ein Projekt zu wählen, das ich schnell fertig stellen kann.

Ich bin auch jedes Mal von den Leistungen der anderen Teilnehmer überrascht und beeindruckt.

Wenn Sie heute eine Stelle in der Spieleindustrie bekommen könnten, was für ein Spiel würden Sie am liebsten entwickeln?

Ich glaube eine Mischung aus den alten Sierra Abenteuerspielen und Siroccos Fenix Blade könnte mich reizen. Eines Tages vielleicht, wenn

mein geheimes Projekt veröffentlicht ist, und DUMB Studio gute Fortschritte macht...

Welche Ratschläge würden Sie einem aufstrebenden Entwickler mit auf den Weg geben wollen?

Nicht aufgeben. Ich habe über 10 Jahre gebraucht, um da zu sein wo ich heute bin. Wenn man seine starken und nicht so guten Seiten kennt, dann ist dies auch ein gewaltiger Vorteil. Nicht jeder kann sowohl programmieren, zeichnen als auch komponieren – wenn man es trotzdem versucht, dann stehen Enttäuschungen ins Haus. Es ist besser, sich anfangs mit »Programmierer Kunst« zufrieden zu geben, und dann einen Zeichner zu suchen, als nach jahrelangen Versuchen, die Grafiken für ein Spiel zu erstellen, entnervt aufzugeben.

Nun, da, wo der eine Schwachstellen hat, da hat ein anderer Stärken. Teamwork ist immer eine lohnenswerte Option.

Richard Phipps

Richard Phipps ist unter anderem der Programmierer von »Dynamite«, einem spaßigen Multiplayer Spiel, in dem man Power-Ups sammelt und Gegner in einem Labyrinth in die Luft jagt. Derzeit arbeitet er an »Chaos Funk«, einem Remake des klassischen Strategiespiels Chaos für den Sinclair Spectrum. Er wurde 2003 von den Lesern des Pixelate Online Magazins zum »vielversprechendsten neuen Programmierer« gewählt.

Was war das erste Computerspiel, das Sie gespielt haben?

Oh! Knifflige Frage... mein Gedächtnis ist ziemlich schlecht, aber ich glaube es könnte eines dieser »Pong« ähnlichen Spiele auf einer der alten Atari-Konsolen gewesen sein. Ich glaube, es gab sogar einen Fußballmodus, in dem man mit 4 Leuten gleichzeitig spielen konnte.

Wie kamen Sie zum Programmieren?

Ich bekam mit 7 einen ZX-81, aber ich glaube nicht, dass ich mit dem Programmieren anfing bevor wir unseren Atari 800 XL hatten. Bis zu diesem Zeitpunkte habe ich vielleicht mal ein paar Zeilen in einem Programm geändert, um in einem Spiel mehr Leben zu erhalten – aber das zählt ja nicht.

Ich fing mit dem Codieren an, weil ich kreativ sein und etwas erschaffen wollte. Es ging mir dabei gar nicht so um Spiele. Ich habe schon immer neben Spielen auch kleine Tools und Routinen geschrieben.

Gibt es irgendein Spiel, das Sie gerne in einer besseren, neuen Version sehen würden?

Es gibt eine Menge Spiele, von denen ich gerne mal neue Versionen schreiben würde. Ich habe ja auch schon zwei Remakes geschrieben. Allerdings gibt es da ein Erlebnis, das bei mir besonders hängen geblieben ist. Als ich deutlich jünger war, bekam ein Freund den brandneuen Atari ST. Als ich auf diesem Computer zum erstenmal das Spiel »North and South« sah, wollte ich sofort eine Version für den Atari 800 schreiben.

Die Idee war eigentlich recht durchgeknallt, vor allem wenn man bedenkt, dass ich ein Spiel, das in einer Auflösung 320x200 bei 16 Farben lief und mit der Maus bedient wurde, auf ein zeichenbasiertes Spiel bei 160 x 120 und 5 Farben und Joystick Bedienung umwandeln wollte. Besonders wenn man in BASIC programmiert und noch immer ein Kind ist (lacht).

Ich glaube ich schaffte es eine ähnlich aussehende Karte, die Nord- und Südcharaktere und einen sich bewegenden Zug hinzubekommen. Da wurde mir erst bewusst, wie schwer die Aufgabe war, die ich mir vorgenommen hatte.

Woher bekommen Sie Ihre Ideen?

Ich habe ständig irgendwelche zufälligen Ideen. Manche Ideen werden durch Bücher und Filme angeregt, andere durch mein Interesse für Philosophie und Psychologie. Allerdings sind die meisten dieser Ideen für den heutigen Markt nicht gerade brauchbar. Der Ideen, die sich wohl am ehesten verwirklichen lassen, sind Kombinationen bekannter Genres oder alte Themen mit einem Twist in eine neue Richtung.

In einigen Fällen bemerke ich erst nach dem Erstellen eines Prototyps, ob sich eine Idee für ein Spiel eignet.

Welche Idee, welches Element eines Spieles hat Sie am meisten beeindruckt?

Nicht einfach zu beantworten. So auf Anhieb fallen mir dazu 3 Dinge ein. Als ich das erste Mal »Dungeon Master« gesehen habe und mir klar wurde, dass man Rollenspiele sehr wohl auch in 3D realisieren kann.

Als ich die ersten Polygone mit Texturen gesehen habe, hat es mich aus den Socken geworfen.

Und schließlich, als mit klar wurde, wie die wenigen Regeln von Dynablaster zu jeder Mengen unterschiedlicher Strategien im eigentlichen Spiel führten, besonders sobald die Kettenreaktionen ins Spiel kommen. Ich denke Dynablaster ist eine der besten Mischungen von Puzzle- und Arcadespielen die ich je gesehen habe.

Sowohl »Dynamite« als auch »Chaos Funk« sind Multiplayer Spiele. Ist dies ein Zufall oder gefallen Ihnen Multiplayer Spiele besser?

Ich habe in meiner Schulzeit mit einigen Freunden beinahe jeden Abend Computerspiele gespielt. Das hat mich ziemlich stark geprägt – Multiplayer Spiele sind einfach unterhaltender als Single-Player Spiele.

Wie ist »Chaos Funk« entstanden? Können Sie uns einen kurzen Überblick über die bisherige Entwicklungsgeschichte geben?

»Chaos Funk« entstand als ich mit meinen Smooth-Zoom-Routinen experimentiert habe. Ich habe mir überlegt, für welche Spiele ich diese Technik einsetzen könnte. »Chaos«, das schon immer zu meinen Lieblingsspielen gezählt hat, ist mir sofort in den Sinn gekommen.

Ich habe einige der Originalgrafiken aus dem Spiel extrahiert und diese als Technologietest für meine Zoom Routinen benutzt. Als ich sah, wie gut diese Grafiken auf dem Schirm aussahen, war mir klar, dass ich weitermachen würde.

Was war der schwierigste Part in der Entwicklung von Chaos Funk?

Ohne Zweifel die künstliche Intelligenz. Das lag zu einem großen Teil daran, dass ich noch nie zuvor etwas in der Richtung programmiert hatte und die Feineinstellung einer AI ein schwieriger Prozess ist. Die Auswirkungen mancher Änderungen sind einfach nicht immer gleich ersichtlich.

Was ist so besonders an Chaos, das Sie sich entschlossen haben ein Remake zu schreiben?

Ich habe früher stundenlang Chaos mit meinen Freunden gespielt. Es erlaubt einem auf verschiedenen Ebenen zu planen – und selbst heute ist es

immer noch ein faszinierendes Spiel. Besonders wenn man sich vor Augen führt, dass dieses Spiel beinahe 20 Jahre alt ist.

Wie schwer war es die textbasierte Menüführung durch eine moderne GUI zu ersetzen?

Nun, die GUI in »Chaos Funk« ist recht einfach gestrickt. Ich kannte die Amiga Version deswegen war mir klar, dass das Auswählen der Zaubersprüche mit der Maus deutlich schneller funktioniert.

Das Hauptproblem war gar nicht die Technik, sondern die Art und Weise wie die GUI mit dem User interagiert. Verschiedene Leute haben unterschiedliche Auffassungen vom User Interface Design. Ideen und Konzepte, für mich selbstverständlich waren, waren für andere nicht nachvollziehbar.

Die jetzige GUI wird vielleicht nicht jedem gefallen, aber ich denke, jeder wird damit zurechtkommen.

Was können Sie uns über die Smooth-Scaling Routinen verraten?

Ahh! (Lacht) Ich wusste, dass diese Frage kommen würde. Wenn die Leute das Spiel zum ersten Mal sehen, erhalte ich viele Kommentare in der Art von: »Das muss ja ewig gedauert haben, all die Grafiken in Vektoren umzuwandeln!«. Allerdings habe ich dies gar nicht getan. Das Spiel wandelt die Grafiken beim Starten automatisch von Bitmaps in Vektorgrafiken.

Die Routinen waren schon von Anfang an sehr schnell. Selbst wenn ich den kompletten Schirm jeden Frame neu vektorisiere bevor ich ihn skaliere, habe ich noch immer 20 Bilder pro Sekunde auf meinem alten 300MHz PC.

Wenn ich bereits fertige Vektorgrafiken nutze, dann gehen die FPS durch die Decke.

Stehen Sie in Kontakt mit dem ursprünglichen Autor von Chaos?

Ich habe bisher noch nicht versucht Jullion Gollop (den Original-Autor) zu kontaktieren. Allerdings kenne ich einige Leute die auch an Remakes gearbeitet haben und sich an ihn gewandt haben. Einige bekamen sehr nützliche Hinweise von ihm, wie bestimmte Teile des Spieles funktionieren.

Ich denke, dass dies ein sehr netter Zug von ihm war, und ich hoffe, dass ihm meine Version gefallen würde.

Chaos Funk ist beinahe fertiggestellt. Was sind Ihre Pläne für zukünftige Projekte?

Es gibt einige Projekte, die ich ernsthaft in Betracht ziehe. Ich glaube ich werde ein Zeichenprogramm schreiben und dabei einige Ideen aufgreifen, die ich in meinem früheren Tool »Style Paint« entwickelt habe.

Allerdings habe ich nun einige deutlich ambitionierte Vorstellungen, und dieses Projekt könnte sich dadurch über einen recht langen Zeitraum erstrecken.

Welche Ratschläge haben Sie für ein neues Mitglied in der Allegro-Gemeinschaft?

Mit Allegro kann man hochauflösende Versionen von beinahe jedem zweidimensionalen Computerspiel erzeugen. Dies schließt alle Spiele für den Amiga, Atari und Konsolen wie das Super Nintendo und Megadrive ein.

Und mit AllegroGL steht einem auch die dritte Dimension offen.

Wenn man sich das vor Augen führt, dann wird klar, dass jeder Entwickler ein gutes Spiel mit Allegro schreiben kann. Vorausgesetzt man hält durch und gibt nicht vorzeitig auf.

Ich spreche hier aus Erfahrung. Dynamite war mein erstes Programm, das ich mit Allegro entwickelt habe, das erste Programm für den PC und auch das erste Programm für das ich C benutzt habe. Ich hatte bis dahin nicht mal ein »Hallo Welt!«-Programm in C geschrieben. Ich habe mir wochenlang die C Syntax aus Beispielen zusammengesucht, bevor ich mir ein C-Buch aus der Bücherei ausgeliehen habe.

Trotzdem habe ich es geschafft einen vollständigen »Bomberman« Clone in etwa 3 Monaten zu schreiben. Als ich mit der Uni fertig war, habe ich Kopien dieses Spieles an etwa 10 Softwarehäuser geschickt und bekam 3 Einladungen zu einem Vorstellungsgespräch, eines davon bei Code Masters – und eine Firma hat mir sogar direkt einen Job angeboten.

Also: Nicht aufgeben – haltet durch, es lohnt sich!

Robert »Bob« Ohannessian

Robert J. Ohannessian ist der Autor von FBLend, einer schnellen Bibliothek zum Überblenden von Bildern. Er ist also einer der treibenden Kräfte hinter DUMB und AllegroGL und Teil der Allegro 5 Development Group.

Sie sind der Allegro Optimierungs-Guru. Ich habe manchmal den Eindruck, sie haben das Timing jedes Assembler Befehls verinnerlicht. Wie wird man so gut im Optimieren von Programmen wie Sie?

Ich bin schon lange davon fasziniert auch den letzten Rest von Performance aus einem Rechner herauszukitzeln. Eventuell liegt es daran, dass ich einen so langsamen Rechner habe (lacht).

Ich habe erst vor ein paar Jahren ernsthaft angefangen mich für Optimierungen zu interessieren, als ich anfing mit dem MMX Befehlen zu arbeiten. Zum gleichen Zeitpunkt fing ich an, einige Routinen in Assembler zu codieren.

Ich habe auch schon immer ein großes Interesse an Grafikprogrammierung gehabt. Also habe ich versucht so viele Echtzeiteffekte wie ich kann in Software zu implementieren. Die meisten dieser Effekte brauchten schnelle Bitmap-Überblend-Routinen.

Da fing ich an, FBlend zu entwickeln. Ich habe ständig das Netz nach Code durchsucht, der das gleiche tat. Wenn ich etwas fand, dann habe ich diesen Code gegen FBlend antreten lassen.

Wenn ich einen neuen, schnelleren Weg fand etwas zu implementieren, dann änderte ich FBlend entsprechend um. Ich habe auch eine Menge von den wahren Gurus der Assembler Optimierung gelesen, insbesondere von Paul Hsieh und Michael Abrash.

Was die Ausführungszeiten der einzelnen Befehle angeht: Ich denke ich kann sie nur deswegen auswendig, weil ich sie so häufig nachgeschlagen habe.

Auf Ihrer Webseite (http://bob.allegronetwork.com/) haben Sie eine Sammlung von Optimierungs-Tricks zusammengestellt. Einige scheinen offensichtlich zu sein, andere sind eher arkaner Natur. Sollte der Compiler nicht von sich aus »a=a/2« in eine Shift-Anweisung umwandeln?

Einige dieser »Tricks« wurden in die neueren Versionen der Compiler übernommen. Der Gnu Compiler wird beispielsweise beinahe jede Division durch eine Multiplikation mit dem Kehrwert ersetzen. Sie sollten allerdings immer einen Blick auf die Ausgabe des Compilers werfen, bevor sie diese Optimierungen anwenden.

Ich denke aber die eigentliche Frage ist: Warum sollte ich mich um die Optimierung kümmern, wenn der Compiler das für mich erledigen kann? Tatsache ist, dass der Compiler eben nicht jeden dieser Tricks selbst anwenden kann. Für die meisten gelten Einschränkungen oder sind nur in Spezialfällen gültig.

Der Programmierer kann einige Umstände voraussetzen oder garantieren, von denen der Compiler keine Ahnung hat. Mit diesen Extrainformationen kann der Code teilweise einfach besser optimiert werden.

Allerdings werden die Compiler immer weiterentwickelt, und auch die Zielmaschinen entwickeln sich weiter. Dadurch holt die Zeit mein kleines Dokument immer schneller ein.

Was sind die normalen Schritte, wenn ich ein Spiel optimieren möchte?

Das Erste was Sie machen sollten, ist ein Tool namens »Profiler« einzusetzen. Dieser wird Ihnen eine Aufstellung liefern, wieviel Zeit in welchem Teil des Programms verbracht wird. Wenn Sie eine Funktion um den Faktor 10 beschleunigen, diese Funktion aber nur 1% der Ablaufzeit belegt, dann haben Sie damit Ihr Programm gerade mal um 0.8% beschleunigt und eine Menge Zeit verschwendet. Werfen Sie lieber zuerst einen Blick auf die Stellen in ihrem Code, in dem das Programm am meisten Zeit verbringt.

Sobald Sie wissen, wofür Sie am meisten Zeit verbrauchen, optimieren Sie diesen Teil des Programms. Dafür haben Sie 2 Möglichkeiten. Entweder machen Sie diesen Teil schneller oder Sie sorgen dafür, dass er nicht mehr so häufig aufgerufen wird.

Sie sollten auch die eingesetzten Algorithmen optimieren. Ersetzen Sie langsame Algorithmen durch schnellere, oder schreiben Sie speziellen Code für häufig gewählte Ausführungspfade. Sobald Sie etwas optimiert haben, sollten Sie sofort den Profiler erneut laufen lassen, damit Sie sicher sein können, dass der Code jetzt nun wirklich schneller und nicht etwa langsamer geworden ist.

Was gibt es von der Allegro-5-Entwicklung zu berichten?

Die größte Änderung ist sicherlich die neue API. Die aktuelle ist etwa 10 Jahre alt und das Alter zeigt sich an manchen Stellen deutlich. Die neue API wurde von Grund auf neu entwickelt. Sie ist konsistenter, leichter, flexibler und leichter auf neue Plattformen zu portieren. Wir versuchen natürlich, wo immer wir können die Einfachheit, der aktuellen API beizubehalten.

Andere Änderungen betreffen den Extra Code, der eigentlich nie richtig zum Allegro-Kern gehört hat. Das betrifft den FLI/ FLC Abspielcode, die GUI, die Software 3D-Engine und die Mathematikroutinen. Diese gehören eigentlich nicht in eine typische 2D Bibliothek und werden aus diesem Grund in Zusatzmodule ausgegliedert.

Andererseits werden Teile bisheriger Zusatzmodule in den Kern von Allegro-5 aufgenommen. FBlend und ein paar Teile von AllegroGL werden beispielsweise mit hoher Wahrscheinlichkeit in den Kern übernommen.

Die neue Regel lautet: »Wenn man es in eine Add-On Bibliothek packen könnte, dann sollte man es auch in eine Add-On Bibliothek packen«. Wir würden gerne Allegro so klein wie möglich halten, um zu verhindern, dass es irgendwann unter dem eigenen Gewicht zusammenbricht. Allegro, wie es heute ist, ist einfach zu groß und sperrig.

Was ist Ihrer Meinung nach das wichtigste neue Feature von Allegro 5?

Hardware-unterstütze Gedankenkontrolle natürlich.

Aber ernsthaft, ich denke die Unterstützung von mehreren Fenstern und Monitoren wird eine der coolsten Features von Allegro 5 werden. Jetzt muss ich nur noch etwas freie Zeit finden, um dieses Feature auch zu implementieren.

Arbeiten Sie derzeit auch an einem Spiel, oder ist die Arbeit an Allegro 5 und den anderen Bibliotheken zu Zeit aufwendig?

Es ist meine reguläre Arbeit, die derzeit am meisten Zeit verbraucht. Sobald ich mal etwas freie Zeit habe, arbeite ich an den Dingen, die ich in diesem Moment für cool halte. An einigen Tagen ist dies Allegro 5, an anderen Tagen Allegro 4, manchmal update ich auch meine Webseite. Ich habe in der jüngsten Zeit leider nicht mehr soviel Zeit an Spielen zu arbeiten.

Welche Spiele sind auf Ihrer Top5 Liste der besten Spiele aller Zeiten? Und was macht diese Spiele so besonders?

Ich würde sagen meine Reihenfolge ist: Chrono Trigger, FreeSpace 2, Final Fantasy VI, StarCraft und Platz 5 teilen sich Super Metroid und Metroid Prime.

Interessanterweise kann ich nicht sagen, was diese Spiele so besonders macht. Ich denke sie schaffen es einfach nur, viele Dinge richtig zu machen. Sie haben tolles Gameplay, einen tollen Soundtrack und einen hohen Wiederspielwert. Die tolle Grafik (nach dem Standard der damaligen Zeit) tut ihr übriges.

Glauben Sie, dass die Spiele der alten Schule ein besseres Gameplay haben als die modernen Spiele?

Für jedes gute Spiel, das damals herauskam, gab es 10 schlechte Spiele. Ich glaube nicht, dass sich an diesem Verhältnis irgendwas bis heute geändert hat.

Was ist Ihr Tipp für neue Nutzer von Allegro?

Habt Spaß! Das ist die Hauptsache dabei.

Johan Peitz

Johan Peitz ist der Kopf von »Free Lunch Design«. Seine Spiele kombinieren witzige Charaktere mit einfallsreichen Spielidee, und haben ihm auf diese Weise zu einem sehr guten Namen unter den unabhängigen Entwicklern verholfen.

Wie sind Sie zum Codieren gekommen?

Ich denke ich habe mit dem Programmieren begonnen als ich so etwa 10 oder 11 Jahre alt war. Ich habe damals nur etwas herumgespielt, aber ich habe dabei eine Menge gelernt. Ich war schon immer ein großer Fan von Brettspielen und ich habe zusammen mit einem Freund ein eigenes entworfen. Als wir dann endlich programmieren konnte, erschien es nur natürlich ein Spiel zu schreiben. Oder anders gesagt: Mein Interesse am Programmieren und am Entwerfen von Spielen waren schon lange da, haben sich eine Zeit lang parallel entwickelt und sind dann verschmolzen.

Gab es ein Spiel, das Sie dazu animiert hat, ein ähnliches zu schreiben?

Als ich gerade anfing etwas besser im Programmieren zu werden, entdeckte ich ein Allegro Spiel namens »Wotan: Raiders of the Sacred Botijo«. Es war ein 2D Weltraum Shooter, der richtig gut aussah. Das einzige Problem war, dass mein Computer für das Spiel zu langsam war. Ich wollte das Spiel aber unbedingt spielen, also fing ich an meine eigene Version davon zu schreiben. So wurde »Operation Spacehog« geboren. Bei diesem Spiel habe ich eine Menge gelernt, es war mein erstes größeres Spiel mit Scripting und jeder menge Grafiken. Ich hatte also jede Menge Spaß beim Schreiben des Spieles.

Wo nehmen Sie Ihre Ideen her?

Ich bekomme die meisten guten Ideen beim Betrachten von Bildern. Das können Photos, Pixel-Arbeiten oder klassische Malerei sein. Manchmal bekomme ich auch gute Ideen von Wörtern. Ich suche mir ein Wort, assoziiere andere Wörter mit diesem oder suche mir ähnlich klingende Wörter; tausche hier und da einen Buchstaben aus. Dadurch entstehen viele interessante Wörter und Wortschöpfungen. Die meisten sind reiner Blödsinn, aber die eine oder andere Kombination von Wörtern kann manchmal einen Funken zünden.

Das Wichtigste ist, außerhalb der gewohnten Bahnen zu denken. Wenn Sie immer über die gleichen Ideen nachgrübeln, dann wird dadurch Ihre Kreativität stark eingeschränkt. Ich versuche jedes Mal ein neues und frisches Spiel zu schreiben, damit ich mich nicht in irgendwelchen Gewohnheiten verstricke.

Wie sammeln Sie Ihre Ideen?

Jede neue Idee schwirrt mir erst einige Zeit im Kopf herum. Früher oder später fange ich dann an Entwürfe der Bildschirme zu zeichnen, mögliche Charaktere und Spielabläufe.

Dies ist auch eine Art Scheidepunkt. Entweder überzeugt mich die Idee und ich realisiere sie in einem Spiel oder ich schreibe einfach alles auf ein Stück Papier und mache erst einmal etwas anderes. Sobald ich es niedergeschrieben habe, brauche ich nur einen Blick auf das Papier zu werfen, um die Idee wieder vor Augen zu haben. Kurz gesagt, ich habe einen großen Stapel mit Papieren daheim.

Ihre Serie von Spielen mit »Alex dem Allegator« ist sehr erfolgreich. Alex4 wurde gerade veröffentlicht. Können Sie uns mehr über Alex erzählen? Wie ist er entstanden? Wie hat er sich entwickelt?

Alex wurde im Rahmen des ersten Speedhack Wettbewerbes erschaffen. Im Speedhack haben die Teilnehmer 72 Stunden Zeit ein Spiel zu schreiben, das einigen, zufällig ausgewählten Regeln entspricht. Ich brauchte einen Hauptcharakter für mein Spiel, und da es zu dieser Zeit auch einige Gespräche über ein offizielles Allegro Maskottchen gab, habe ich Alex erschaffen. Alex, Allegator, Allegro.... es war beinahe zu einfach (lacht). Im ersten Spiel war er einfach ein normaler Alligator auf der Suche nach seiner Freundin Lola, welche im Dschungel spurlos verschwunden war.

Das Spiel konnte im Speedhack Wochenende leider nicht fertiggestellt werden, trotzdem bekam es gute Wertungen und die Spieler mochten des Konzept von Alex. Ich beschloss Alex zu behalten, und verwendete die Figur auch in den darauf folgenden Speedhacks.

Das zweite Spiel mit Alex wurde ein Puzzle (das war eine Speedhack Regel) und beinhaltete nicht nur Alex, sondern auch seinen Bruder Aaron, die beide im Spiel antraten.

Alex III ist ein Rennspiel, das auch nicht beendet werden konnte. Auf den ungeraden Nummern scheint ein Fluch zu liegen (lacht).

Für einen anderen Wettbewerb programmierte ich dann Alex IV, das wieder zu den Hüpfspielwurzeln von AlexI zurückkehrt.

Alex war schon immer ein sehr witziger Charakter, der stets bereit ist anderen zu helfen. Er ist ein Held und hat keine Angst vor Gefahren. Über die verschiedenen Spiele hinweg hat er sich von einem hässlichen Alligator zu einem süßen, liebenswerten Helden entwickelt. Ich will nicht das Ende von Alex IV vorwegnehmen, aber sagen wir mal, es endet mit einem netten Cliffhanger. Dies war noch nicht das letzte Spiel mit Alex.

Was ist die Geschichte hinter Ihrem ersten grossen Hit, »Happyland Adventures«?

Auch dieses Spiel wurde für einen Wettbewerb geschrieben. Diesesmal ging es darum in ein paar Wochen ein Spiel zu schreiben, welches einen vorgegebenen Satz von Grafiken benutzt. Die meisten dieser Grafiken hatten ein »Space Invaders« Feeling. Da ich gerade die Arbeiten an »Operation Spacehog« beendet hatte, war mir nicht mehr so nach Weltraum.

Die übrigen Grafiken waren ein sommerlich wirkenden Satz von Tiles und einige kleine Kreaturen. Ich habe all dies in die Luft geworfen, und das Happyland Konzept kam wieder nach unten. Ich habe aus dem kleinen Getier die hilflosen Happylander gemacht und nutzte andere Charaktere für den Helden und die Gegner. Da ich keinen normalen Plattformer machen wollte, und die Sprites, die es gab nicht wirklich mit Pistolen und Schüssen funktionierten, habe ich beschlossen ein friedliches »Sammel-Sie-Alle-Ein«-Spiel zu machen.

Kinder lieben es Dinge zu sammeln und ich glaube Happyland reitet auf der gleichen Welle wie Pokémon. Sammele sie alle, und du gewinnst. Als ich den Karteneditor für Happyland veröffentlichte, fingen die Leute an eigene Karten zu erstellen und über das Internet zu verteilen. Es gibt jetzt Hunderte von Karten im Internet, die man sich herunterladen kann.

Das Spiel »Icy Towers« ist ein weiterer Hit. Es gibt bereits Fansites und Foren, die sich mit dem Spiel befassen. Haben Sie mit einer solchen Resonanz gerechnet?

Auf keinen Fall. Ich hielt es nur für ein kleines Spiel, als wir es veröffentlicht haben. Wir hätten nicht erwartet, dass es sich wie ein Buschfeuer verbreiten würde. Menschen jeden Alters überall in der Welt haben es gespielt und ihre Punktzahlen verglichen.

Der Held von »Icy Towers«, Herold the Homeboy, wird bereits in einem Atemzug mit berühmten Spieliguren wie Mario und Sonic genannt. Wie entstand dieser Charakter?

Wieder einmal ist das ganze Konzept im Rahmen eines Wettbewerbs entstanden. Ich brauchte einen Charakter, und da das ganze Spiel eher abstrakt war, wollte ich einen farbenfrohen Kontrast zum Rest des Spieles. Herold wurde geboren, er ist eine Figur, die wahrscheinlich niemals einen solchen Turm betreten würde, aber als er es tat, hat es ihm gefallen.

Im Nachhinein würde ich sagen, dass es den meisten Leuten mit diesem Spiel ähnlich ergeht. Am Anfang denken Sie nicht, dass das Spiel wirklich Spaß machen könnte, aber wenn Sie einmal anfangen zu spielen, dann lässt es sie nicht mehr los.

Die Soundeffekte in Ihren neueren Spielen helfen wirklich den Spieler in das Geschehen hineinzuziehen. Wie wichtig sind Soundeffekte und die Musik eines Spieles für Sie?

Sowohl Sound als auch Musik sind sehr wichtig. Ich habe allerdings kein Talent auf diesem Gebiet, wenn ich also Sounds oder Musik bastele, dann ist das schwere Arbeit für mich – und die Chance, dass dabei etwas Gutes herauskommt sind eher gering. Glücklicherweise habe ich Anders Svenson in meinem Team, der auf diesem Gebiet extrem talentiert ist. Er hat alle Effekte für Icy Towers und Alex IV beigesteuert, und ich glaube, dass das Ergebnis wirklich großartig geworden ist. Wir sprechen normalerweise etwas über die Stimmung und die Umgebung des Spieles, dann bekommt Anders freie Hand und überrascht mich dann mit den Ergebnissen (lacht).

Gibt es irgend etwas, das Sie gerne an einem Ihrer bisherigen Spiele ändern würden?

Ich glaube die Spiele sind vom Gameplay her genau so geworden, wie ich sie mir vorgestellt habe. Natürlich bekommt man beim Programmieren neue Ideen und würde gerne tonnenweise neue Features hinzufügen – aber ich denke, dass es besser ist, die Spiele einfach und simpel zu halten und dabei so nah es geht bei der ursprünglichen Idee zu bleiben. Gute Ideen kann man jederzeit auch in späteren Spielen verwenden – es gibt keinen Grund sie alle in einem Spiel zu implementieren.

Alle Ihre Spiele wirken sehr professionell. Können Sie ein paar Tipps verraten, wie man einen solchen Eindruck erzielt?

Dieser Eindruck entsteht nicht von alleine. Es ist harte Arbeit. Wenn das eigentliche Spiel beendet ist, dann liegen noch etliche Stunden vor einem, in denen man die kleinen Ecken und Kanten ausbügeln muss. Die meiste Zeit sind es wirklich nur Kleinigkeiten, und wenn ein Spiel den Endruck vermittelt »nicht ganz fertig« zu sein, dann liegt das meist eher daran, dass die Entwickler ihr Spiel zu schnell der Öffentlichkeit zeigen wollen.

Achten Sie darauf, dass Ihr Spiel einheitlich ist. Benutzen Sie keinen Science Fiction Zeichensatz wenn Ihr Spiel im Mittelalter handelt. Wenn Sie einen Font oder eine Grafik benutzen die antialiased ist, dann sollten sie entweder alle anderen Grafiken auch antialiasen oder dafür sorgen, dass keine Grafik glatte Kanten hat. Stellen Sie sicher, dass alle Bildschirmüberblendungen korrekt arbeiten. Es sind wirklich die kleinen Dinge, die einen großen Unterschied machen.

Wie wichtig ist eine Webseite Ihrer Meinung nach?

Wenn Sie professionell erscheinen wollen, dann brauchen Sie eine gute Website. Sie muss nicht mit den letzten Plug-Ins vollgestopft sein. Es reicht, wenn die Webseite solide und in sich stimmig ist. Das ist genauso wie bei den Spielen (lacht).

Was sind Ihre nächsten Projekte?

Alex IV ist gerade veröffentlicht worden und ein Update zu Icy Towers ist auf dem Weg. Wenn diese beiden Spiele abgeschlossen sind, habe ich Zeit mir etwas neues zu überlegen. Ich habe da eine Handvoll Ideen, die ich wirklich gerne weiter verfolgen würde.

Ich könnte mir sehr gut vorstellen einen Nachfolger von Happyland zu schreiben, ich denke es gibt auf diesem Gebiet Einiges, was man noch probieren könnte. Alle meine Ideen sind derzeit auf eine 2D Umgebung ausgelegt, also ist mein nächster Plan erst mal eine wirklich flexible 2D Engine zu schreiben, die all das verrückte Zeug machen kann, das ich vorhabe.

Welchen Tipp würden Sie einem Allegro Anfänger mit auf den Weg geben wollen?

Als ich mit dem Entwickeln von Spielen anfing, habe ich mir immer all die riesigen Spiele ausgemalt, die ich machen wollte. Allerdings kam es da nie dazu, weil ich einfach nicht wusste wie. Ich glaube es ist immer gut erst mal klein anzufangen, und nicht mehr abzubeißen als man kauen kann. Machen Sie kleine Schritte auf dem Weg zu Ihrem großen Ziel.

Anhang

Im Anhang finden Sie ein Kapitel zur Installation von Allegro sowie ein ausführliches Glossar. Anhang C informiert Sie über den Inhalt der Buch-CD.

A Allegro installieren

Hier erfahren Sie Wissenswertes über das Installieren von Allegro.

Benötigte Files

Unter Windows benötigen Sie einen C/ C++ Compiler, empfohlen wird hier die GNU Compiler Collection (GCC). Sie bekommen eine Version von MinGW (Minimalist GNU für Windows) unter *http://www.mingw.org/*.

Überprüfen Sie nach dem Download und der Installation unbedingt das `\bin` Verzeichnis des Compilers (zum Beispiel `c:\mingw32\bin`). In diesem Verzeichnis muss eine Datei namens `make.exe` liegen. Ist diese Datei nicht vorhanden, dann erstellen Sie eine Kopie der Datei `mingw32-make.exe` und benennen Sie diese Kopie dann in `make.exe` um.

Unter Linux ist in der Regel bereits ein Compiler installiert.

Zusätzlich zu einem Compiler wird auch noch die Allegro Distribution benötigt und unter Windows das „kleine" DirectX SDK.

Sie bekommen jede dieser Dateien auf der Allegro Homepage: *http://alleg.sourceforge.net/wip.html*.

Sie brauchen auf alle Fälle eine Version von Allegro (derzeit `all403.zip` (stable) oder `all4111.zip` (development) für Windows Nutzer), bzw. `allegro-4.0.3.tar.gz` (stable) und `allegro-4.11.1.tar.gz` (development) für Nutzer von Linux Systemen.

Entpacken Sie Allegro in ein beliebiges Verzeichnis, zum Beispiel `c:\allegro\`.

Window Users müssen nun noch das Mini DirectX SDK herunterladen. Dieses finden Sie auch auf der Allegro Homepage, auf der Downloadseite unter „Miscellaneous Files". Für die DirectX-8 Version laden Sie bitte *dx80_mgw.zip* herunter und extrahieren Sie das File in Ihr MinGW Verzeichnis.

Installation

Öffnen Sie nun eine Kommandozeile (Windows) oder Shell (Linux) und wechseln Sie in das Allegro Verzeichnis.

Windows User nutzen führen nun bitte folgende Befehle aus:

```
fix mingw32
make
make install
```

Dadurch wird Allegro auf Ihrem Rechner installiert.

Auf einem Linux System ist die Vorgehensweise ähnlich. Geben Sie folgendes in der Shell ein:

```
chmod +x fix.sh
./fix.sh unix
./configure
make
su -c "make install"
su -c "make install-man"
```

Und das war es. Allegro ist nun auf Ihrem Rechner installiert.

Sollte es bei der Installation zu Problemen gekommen sein, sollten Sie wie folgt vorgehen:

Lesen Sie die Ihrem System entsprechende Datei im Verzeichnis `allegro/docs/build`. Also zum Beispiel `allegro/docs/build/mingw32.txt`. Diese Datei enthält die aktuellsten Informationen zur Allegro Installation.

Sollte Ihnen dies nicht weiterhelfen, besuchen Sie die Allegro Community auf *http://www.allegro.cc/* und beschreiben Sie Ihr Problem im „Installation, Setup & Configuration"-Forum. In der Regel wird hier sehr schnell geantwortet.

B Glossar

Abenteuerspiel
Abenteuerspiele basieren in der Regel sehr stark auf dem Lösen von Rätseln und dem Erkunden eines in der Regel recht komplexen Handlungsablaufs. Beispiele für dieses Genre sind Spiele wie »Space Quest« und die »Monkey Island«- Serie.

Allegro
Eine für viele unterschiedliche Plattformen erhältliche Bibliothek zur Spieleentwicklung. Kombiniert Eingabe-, Sound-, Grafik- und Musikroutinen.

API
Application Programming Interface. Eine Sammlung von Routinen, die als Schnittstelle zu einem bestimmten System dienen. DirectX ist eine Windows API, Allegro ist eine plattformübergreifende API zur Spieleentwicklung.

Artificial Intelligence
Siehe: Künstliche Intelligenz.

Attribut
In einem Rollenspiel sind Attribute Zahlen, die bestimmte Aspekte eines Charakters repräsentieren. Beispiele sind: Stärke, Intelligenz, Glück, Charisma und so weiter.

Bar
Die zweite metasyntaktische Variable. Siehe: Foo.

BASIC
Beginners All-Purpose Symbloc Instruction Code. Programmiersprache, die ursprünglich entwickelt wurde, um die Grundlagen der Programmierung zu vermitteln.

Benutzerschnittstelle
Die Schnittstelle, durch die der User mit dem Programm kommuniziert. Dies kann sowohl ein grafisches Menü System sein, die Tastenkommandos die das Spiel unterstützt oder die Art und Weise wie das Spiel dem Nutzer Informationen übermittelt. Sobald ein Austausch von Informatio-

nen zwischen dem Nutzer und dem Programm stattfindet, ist dies Teil der Benutzungsschnittstelle.

Blit

Block Image Transfer: Das Kopieren eines (Bild)Speicherbereichs in einen anderen.

Boss

Normalerweise ein starker Gegner am Ende eines Levels. Er folgt häufig einem bestimmten Muster, das der Spieler erkennen und lernen muss, wenn er den Boss besiegen möchte.

Clone

Zum einen Spiele, die versuchen bestimmte Aspekte eines anderen Spiels möglichst genau zu imitieren, oder auch Spiele, welche die Idee eines anderen Spieles kopieren. Beispiel: Es gibt sehr viele »Pong«- und »Space-Invader«-Clones im Internet.

Cutscene

Siehe: Zwischensequenz.

Design Dokument

Ein Dokument in dem alle wichtigen Punkte des Spieles aufgelistet sind. Das Design Dokument wird manchmal auch als »Design Bibel« oder einfach nur »Bibel« bezeichnet.

DirectX

Eine Sammlung von APIs (siehe dort), die von Microsoft entwickelt wurde, um Entwicklern einen schnelleren Zugriff auf Hardware Features zu ermöglichen. Die einzelnen APIs beginnen in der Regel mit Direct, wie zum Beispiel DirectInput, DirectDraw etc.

Emulation

Die Simulation einer bestimmten Umgebung. Es gibt Programme, die alte Computer Systeme emulieren. So könnten Sie klassische Computerspiele auf einem C64 Emulator spielen.

Engine

Ein Teilprogramm mit einer bestimmten Aufgabe. Die Grafik Engine sorgt dafür, dass die Grafik sauber angezeigt wird, die Sound Engine kümmert sich um Geräusche etc.

Foo

Die erste metasyntaktische Variable. Wann immer ein Entwickler einen beliebigen Namen für etwas braucht, wird er `foo` verwenden: »Angenommen Sie haben eine Funktion `foo()` mit folgenden Eigenschaften ...«.

Siehe auch: Bar.

Game Engine

Teilsystem, das sich mit der Spielsteuerug und der Koordination seiner Untersysteme beschäftigt. Zu einer Game Engine gehören in der Regel eine Sound, Grafik, Physik und AI Engine.

Interpreter

Ein Programm, das andere Programme ausführt. Der Unterschied zu einem Compiler besteht darin, dass der Compiler das Programm in ausführbaren Code übersetzt, wohingegen der Interpreter jede Anweisung des Programms zur Laufzeit ausführt.

Künstliche Intelligenz

Der Versuch, ein Programm intelligent erscheinen zu lassen.

Labyrinth

Ein Irrgarten aus verschlungenen Wegen. Der Spieler muss sowohl mit seiner Umgebung als auch mit seinen Gegnern zurecht kommen.

Maze

Siehe: Labyrinth.

NPC

Non-Player-Character oder auch Nicht-Spieler-Charakter. Eine vom Computer kontrollierte und gespielte Person.

Pathfinding

Siehe: Wegfindung

Power-up

Eine Belohnung für den Spieler, die diesem in der Regel für eine kurze Zeit bessere Eigenschaften verleiht. In Pac-Man sind die in den Ecken platzierten großen Punkte Power-Ups, die es dem Spieler erlauben Jagd auf die Geister zu machen.

Rekursion
Wenn eine Funktion sich zur Lösung einer bestimmten Aufgabe selbst aufruft, spricht man von Rekursion.

Role Playing Game
Siehe: Rollenspiel.

Rollenspiel
Simulation eines oder mehrerer Helden, die Abenteuer bestehen müssen. Charakteristisch ist die Beschreibung der Spielfiguren durch Attribute (siehe dort) und die Möglichkeit seine Spielfigur durch häufiges Spielen zu verbessern.

Scripting
Der Einsatz einer Scriptsprache. Scriptsprachen sind in der Regel interpretierte Sprachen. Durch Skripte können bestimmte Probleme auf einer höheren Abstraktionsebene behandelt werden.

SDK
***S**oftware **D**evelopment **K**it*

Standard Template Library
Die STL besteht aus einer Reihe von Template-Klassen, die als Teil des Compilers vertrieben werden. Sie stellen Container wie Listen und Stacks zur Verfügung.

STL
Siehe: Standard Template Library.

User Interface
Siehe: Benutzerschnittstelle.

Wegfindung
Der Vorgang, einen möglichst idealen Weg zwischen zwei Punkten zu berechnen.

Zwischensequenz
Eine Animation, die in bestimmten Situationen die Handlung vorantreibt oder ein Ereignis darstellt.

C Inhalt der Buch-CD

Die beiliegende Buch-CD enthält die Beispielprogramme aus dem Buch und die meisten im Buch beschriebenen Tools.

Compiler

In diesem Verzeichnis finden Sie den Gnu C/C++ Compiler, genauer gesagt den Minimalist GNU for Windows (MinGW) in der Version 3.2.3.

Dia_Unix

In diesem Ordner finden Sie die Unix-Variante des Diagramm-Editors.

Mit Dia können Sie Diagramme aller Art erzeugen, zum Beispiel Klassendiagramme und Sequenzdiagramme.

Dia_Windows

Dieser Ordner enthält die Windows-Variante von Dia.

Gimp_Unix

In diesem Ordner finden Sie die Unix-Variante des Bildbearbeitungsprogramms Gimp.

Gimp verbirgt eine erstaunliche Anzahl von Funktionen unter einer etwas gewöhnungsbedürftigen Oberfläche. Unter Unix stehen Ihnen mehr Skripte zur Verfügung.

Gimp_Windows

In diesem Ordner finden Sie die Windows-Variante des Bildbearbeitungsprogramms Gimp.

Libs

In diesem Verzeichnis finden Sie die Tools Allegro, FBlend und DUMB.

Mappy_Unix

Dieser Ordner beinhaltet die Unix-Variante des Map-Editors Mappy.

Mappy_Windows

In diesem Ordner finden Sie die Windows-Variante des Map-Editors Mappy.

Src

Dieser Ordner beinhaltet die Quellcodes der im Buch behandelten Beispiele.

Tools

In diesem Ordner finden Sie die Entwickler-Tools MkExpl, TTF2PCX und DLG (Allegro Dialog Editor).

Winzip

In diesem Ordner finden Sie eine zeitlich begrenzte Shareware-Testversion des Komprimierungstools WinZip 8.1, mit der Sie Dateien zippen und gezippte Dateien entpacken können. Zur Installation doppelklicken Sie einfach auf die exe-Datei und folgen Sie den weiteren Anweisungen.

Nähere Informationen zu den Tools finden Sie an den entsprechenden Stellen im Buch.

Index

A
A*-Algorithmus 687
Active Time Battles 115
Additionssystem 91
Algorithmen 174
Allegro 195, 201, 314, 755
  Alternativen 199
  Benötigte Files 755
  Bewegung 209
  Bilder laden und anzeigen 206
  Doublebuffering 212
  Funktionalitäten 196
  Grafikmodi 202
  Halbtransparenzfunktion 653
  installieren 756
  Konstante Geschwindigkeit 215
  Sound 213
Angriff 571
Animationen 301, 378, 599
Archetypen 101
Attribute 94
Ausgang 603, 608

B
Balance 67, 69
BeatBox 439
Berserker 618
Bewegung 209
Bilder
  laden und anzeigen 206
Bildschirme 262
Bitmap 361
Blitter Object 361
Blutmagier 617
Bob 361
Bogenschütze 615
Bounding Rectangles 390
Buttons 320

C
C 131, 149
C++ 149
  Funktionen 149
  Häufige Probleme 162
  Klassen 154
  Objektorientierte Programmierung 154
  Polymorphie 158
  Strukturen 152
Charaktere 97, 524, 611, 619
  Bogenschütze 615
  Krieger 611
  Magier 613
ClanLib 199
Compiler 133
Container 172
Crossfades 675

D
Deque 172, 182
Design-Dokument 121, 261
Dialog Editor 315
Dialoge 299, 315
Dialogfunktionen 318
DirectX 195
Disbalancen 69
DLG 315
DND3E 199
Doublebuffering 212

E
Editoren 505
Effekte 641
  Explosionen 650
  FBlend 659
  Licht und Transparenz 653
  Lichteffekte 656
  Partikelsysteme 641

Raketentriebwerk 647
Ein- und Ausblendungen 663
Erfahrungspunkte 99, 619
Explosionen 650

F
Fähigkeitsbäume 95
Farben 458
FBlend 659
Fehler 264
Fertigkeiten 94
Fertigkeitsnetze 96
Finite State Machine 276
Fonts 290, 291, 292, 295
Freyr 51
Functor 188
Funktionen 149

G
Game Over 264
gcc 131
GDI 195
Gegenspieler 543
Gegner 543
Geräusche 428
verändern 430
Geschichten 53
Geschosse 589
Geschwindigkeit 215
Gestaltwandler 618
Gleichgewicht 67
Grabber 292
Grafiken 361, 445
Grundgerüst 265
GUI-Elemente 314
GUI-Routinen 316

H
Hauptmenü 299
Heimdall 51
Heldenreise 53
Herakles 46
Aufgaben des 46

I
Ideen 53
Entwicklung von 35
finden 35, 44, 45, 49
Kreativitätstechniken 38
Installationsprogramm 463
Interaktion 523
Direkte Kontrolle 222
Indirekte Kontrolle 223
Mittelbare Kontrolle 222
Modelle 221
Isometrie 625
Isometrische Karten 625
Geschichte 626
Implementierung 628
Iteratoren 173

J
Joystick
Buttons 247
Knüppelinformation 247
Setup 244

K
Kacheln 474
animierte 599
Kampf 571
Echtzeit 112
rundenbasiert 114
Kampfregeln 571
Kampfsystem 105
Karten
Editoren 505
isometrische 625
komplex 499
Klassen 154, 611, 617
Kollision 389
Kollisionsabfrage 389, 393
Bounding Rectangles 390
Kreativitätstechniken 38, 43
6-Farben-Denken 41
6W-Technik 38
Brainstorming 40
Morphologischer Kasten 43

Walt-Disney-Strategie 42
Krieger 611
Künstliche Intelligenz 555

L
Level 619
Anstieg 611
wechseln 603
Level Design 72
Lichteffekte 653, 656
Linker 134
List 172
Listen 178, 181, 182
Look and Feel 261
Lua 707

M
Magie 587
Geschosse 589
Objekte 596
Magier 613
Magiesystem 107
Make 137
Makefiles 137
Muster-Regeln 139
Makefile 270
Map 173, 184
Map-Klassen 479
Mappy 505
Maus
Knöpfe 241
Mauszeiger 237
Radposition 241
Setup 235
Mausunterstützung 302
Medusa 48
Menülisten 303
Menüs 299
Animation 301
Farben 300
Größe 300
Implementierung 303
Marker 301
Mausunterstützung 302
Menülisten 303
Sprachausgabe 302
Minispiele 85
Casino- und Glücksspiele 85
Quizspiel 87
Sportspiele 86
Monster 543
Multimap 173
Multiset 173
Musik 433
Mythen, Sagen und Märchen 45

N
Nordische Sagen 49

O
Objekte 523, 532, 596
Objektorientierte Programmierung 154
Odin 49
Odysseus 46
Osbournes Checkliste 44

P
Panzer 617
Parallax Scrolling 415
Parameterübergabe 172
Partikelsysteme 641
Perseus 48
Perspektive 445
Draufsicht 446
Fluchtpunktperspektive 451
Isometrische Perspektive 449
Kavalierperspektive 449
Klang-Perspektive 448
Seitenansicht 447
Vogelperspektive 451
Zentralperspektive 450
Planung 261
PLib 200
Polymorphie 158
Preprozessor 132
Priester 617

Priority Queue 172
Programmierung 131
objektorientiert 149, 154
prozedural 149
Puzzle 75
abstrakte Rätsel 82
Kistenschieben 76
klassisches Rätsel 82
logische Rätsel 83
Musik- und Rhythmus 83
Objektkombinations- 81
physikalische 76
Schalterrätsel 78
Teleporter-Puzzle 79
Türme von Hannoi 79

Q
Queue 172, 182
Quizspiel 329
Bestenliste 331, 353
Bildschirm 332
Button 339
Fragen 342, 348
Hauptmenü 331, 341
Hauptprogramm 339
Implementierung 335
Information 332
Planung 329
Quizscreen 337
Soundeffekte 333
Spielablauf 350

R
Raketentriebwerk 647
Rogue 473
Rollenspiel 89, 90, 473
Attribute 94
Begriff 89
Charakter 97
Entwurf 91
Erfahrungspunkte 99
Fähigkeitsbäume 95
Fertigkeiten 94
Handel 116

S
Sammelsystem 93
Schlüsselsatz 262
Schurke 618
Schuss-Implementierung 589
Screens 262
Scripting 707
Scrolling 403, 405, 415, 499
Seitenübergänge 661
Set 173
Simple Direct Media Layer 200
Sound 213, 427, 433
Bewegung 429
Kampfgeräusche 429
selber aufnehmen 428
Umgebungsgeräusche 430
Soundeffekte 427
Spezialeffekte 587, 641
Spielablauf 275
Spielbarkeit 67
Spielbeschreibung 262
Spiele
Definition 21
Kategorien 22
veröffentlichen 461
Spielende 264
Spieler
Arten 29
Spielfeld 263
Spielfiguren 485
Splitscreen-Effekt 408
Sprachausgabe 302
Sprites 361, 378, 389, 485, 543
Arten 361
Bitmap 361
Compiled Sprites 368
erzeugen 368
RLE-Sprites 367
Stack 172
Standard Template Library 171
Algorithmen 174
Container 172
Deques und Queues 182
Functor 188

Grundlagen 171
Iteratoren 173
Listen 178
Maps 184
Parameterübergabe 172
Schnelligkeit 171
Vektoren 174
Zeiger 187
Sternensystem 641, 645
Strukturen 152

T
Tastatur
Setup 223
Tastaturpuffer 229
Textausgabe 285
Funktionen 285
Textcodierung 289
textout()-Funktion 285
textprint()-Funktion 287
Thor 50
Tile Animation 599
Tile Based Maps 473
Tile Map 499
TileFAQ 514
Tiles 474
Titelbild 263, 271, 445
entwerfen 452
Transparenzeffekte 653
Treffer 580
TTF2PCX 291

U
Übergänge 661

V
Vampir 617
Vampirgeschichte 58
Vektoren 174, 176, 177
Veröffentlichung von Spielen 461
Bekanntmachung 469
Checkliste 461
Installationsprogramm 463
Website 468
Virtuelle Kamera 405

W
Website 468
Wegfindung 687
Wegfindungsalgorithmen 687

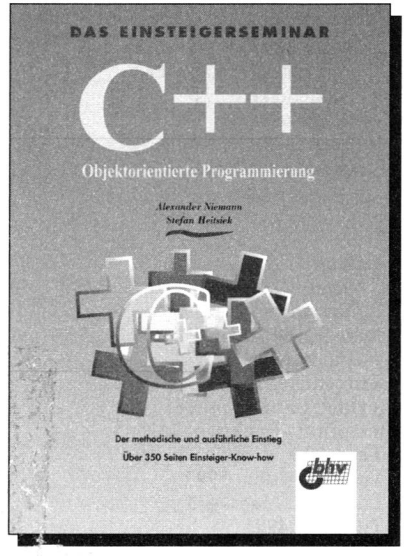
Z
Zeichensätze
farbig 296
Zeiger 140, 187

DAS EINSTEIGERSEMINAR

C++

Objektorientierte Programmierung

Alexander Niemann
Stefan Heitsiek



Der methodische und ausführliche Einstieg
384 Seiten
Einsteiger-Know-how

C++ ist eine objektorientierte Programmiersprache, die auf Basis der Programmiersprache C entwickelt wurde. Obwohl C++ – verglichen mit anderen Sprachen wie bspw. Java – eine ältere Programmiersprache ist, erfreut sie sich nach wie vor großer Beliebtheit. Kaum ein Programmierer kommt im Laufe seines Berufslebens um C++ herum und für jeden Informatikstudenten ist diese Sprache obligatorisch. Profundes Wissen über C++ bildet noch immer die beste Grundlage, um andere Programmiersprachen zu erlernen. Dieses Einsteigerseminar führt in die Programmierung mit C++ und in die Denkweise der objektorientierten Programmierung ein. Dabei werden keinerlei Programmierkenntnisse vorausgesetzt. Mit zahlreichen Programmbeispielen und Übungsaufgaben können Sie die erlernte Theorie direkt in die Praxis umsetzen und gelangen schnell zu ersten Ergebnissen. C++ und dieses Buch bieten die optimale Grundlage für den erfolgreichen Einstieg in die Welt der Programmierung.

ISBN 3-8266-7188-0

* unverbindl. Preisempf.

(D) € 9,95
(A) € 10,30*

verlag moderne industrie Buch AG & Co. KG • Königswinterer Straße 418 • 53227 Bonn • Fax: 02 28 / 970 24 21 • http://www.vmi-Buch.de

DAS bhv TASCHENBUCH: DIE PRE

momox
momox.com/sale
A-6qt8tk

Spiele-programmierung

DAS bhv TASCHENBUCH

- Effektiver Einsatz des B
  Gliederung in fünf Teile
- Erfolgreicher Einstieg in
  und in sich abgeschlosse
- Schnelles und problemlo
  mationen durch breit ang
- Beiliegende CD-ROM mit der beliebten Spieleprogrammierungs-Bibliothek Allegro, den Beispiel-Quelltexten aus dem Buch sowie weiteren nützlichen Tools

Spiele faszinieren Menschen seit jeher. Jahr für Jahr kommen unzählige neue Spiele auf den Markt und finden begeisterte Anhänger. Nicht anders ist es im Bereich der Computerspiele. Wenn auch Sie ein Spiele-Freak sind und einfach einmal wissen möchten, wie Computerspiele entstehen, oder selbst ein Spiel programmieren wollen, sind Sie mit diesem Buch bestens beraten. Praxisnah und leicht verständlich werden sowohl Anfänger-Fragen als auch anspruchsvollere Themen behandelt. Sie erfahren alles, was Sie wissen müssen, um Computerspiele zu programmieren: von der Ideenfindung und der Entwicklung von Geschichten bis zum Entwurf und der Implementierung eines eigenen Spiels. Ausführungen zu fortgeschrittenen Techniken wie grafischen Spezialeffekten, Skriptsprachen, Perspektive etc. runden das Buch ab.

TEIL I: DER DESIGN-PROZESS
Grundlagen; Entwicklung von Ideen und Geschichten; Spielbarkeit und Balance; Rollenspieldesign; Das Design-Dokument

TEIL II: EASY CODING
C- und C++-Grundlagen; Standard Template Library; Spieleprogrammierung mit Allegro; Interaktion; Textausgabe; Menüs und Dialoge; Quizspiele; Sprites; Kollisionsabfrage; Scrolling; Soundeffekte; Titelbilder und Grafiken; Veröffentlichung von Spielen

TEIL III: ROLLENSPIELE
Tile Based Maps; Spieler-kontrollierte Sprites; Komplexere Karten; Objekte; Gegenspieler; Kampf; Spezialeffekte; Krieger, Magier und Bogenschütze

TEIL IV: WEITERFÜHRENDE METHODEN
Isometrische Karten; Effekte; Seitenübergänge; Wegfindung mit A*; Scripting mit Lua, Entwickler-Interviews

TEIL V: ANHANG
Installation von Allegro; Glossar; Inhalt der Buch-CD

ISBN 3-8266-8075-8
(D) € 16,95
(A) € 17,50*
*unverbindliche Preisempfehlung
9 783826 680755